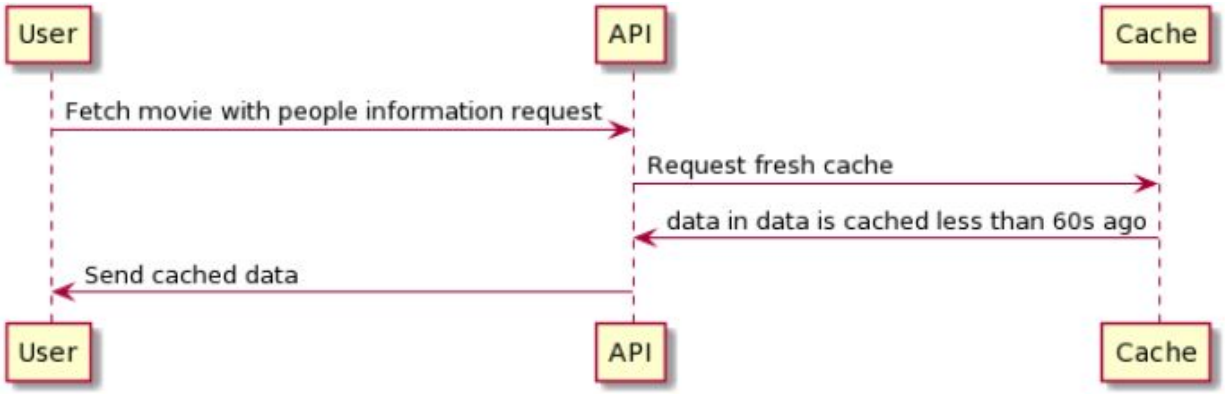
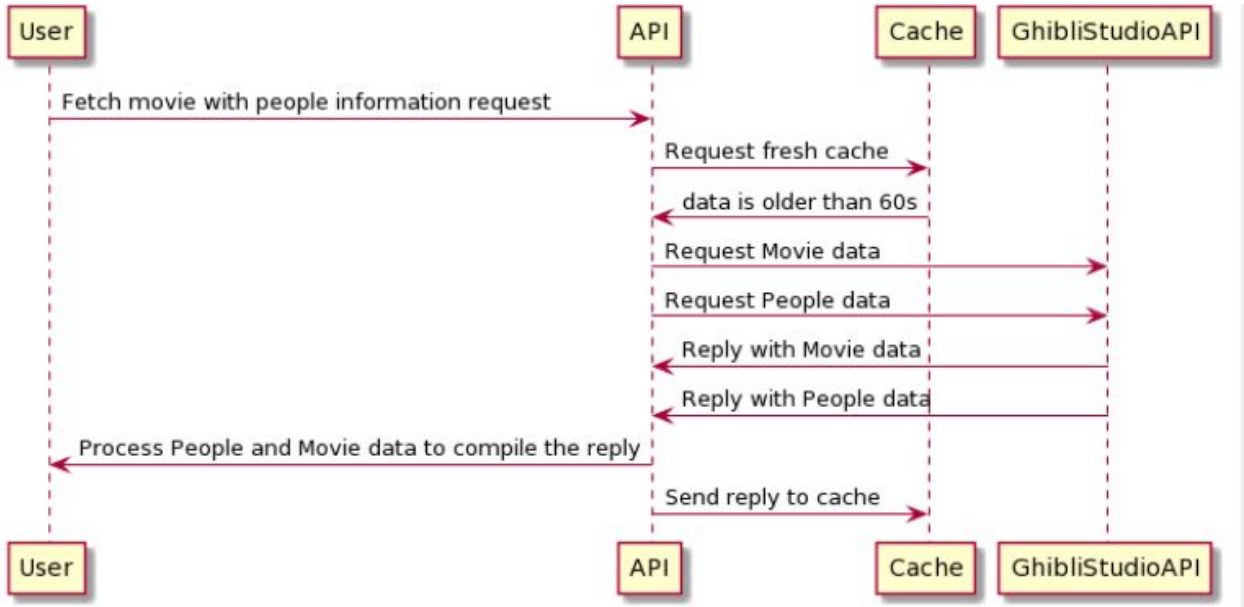
Explanation on the approach.

The workflow of the request in case the reply is already cached and the cache is less than 60s old is presented below



The workflow of the request in case the reply is not yet cached or cache is older than 60s old is presented below



In both cases, only 2 requests maximum that must be sent to GhibliStudioAPI.
Note the format of the reply from GhibliStudioAPI

film	{
	"id": "2baf70d1-42bb-4437-b551-e5fed5a87abe",
	"title": "Castle in the Sky",
	"description": "The orphan Sheeta inherited a mysterious crystal that links her to the mythical sky-kingdom of Laputa.,
	"director": "Hayao Miyazaki",
	"producer": "Isao Takahata",
	"release_date": "1986",
	"rt_score": "95",
	"people": [
	"https://ghibliapi.herokuapp.com/people/"
	],
	"species": [
	
	"https://ghibliapi.herokuapp.com/species/af3910a6-429f-4c74-9ad5-dfe1c4aa04f2"
	],
	"locations": [
	"https://ghibliapi.herokuapp.com/locations/"
	],
	"vehicles": [
	"https://ghibliapi.herokuapp.com/vehicles/"
	],
	"url": "https://ghibliapi.herokuapp.com/films/2baf70d1-42bb-4437-b551-e5fed5a87abe"
	}
		Each film has a unique ID to identify

person	{
	"id": "ebe40383-aad2-4208-90ab-698f00c581ab",
	"name": "San",
	"gender": "Female",
	"age": "17",
	"eye_color": "Brown",
	"hair_color": "Brown",
	"films": [
	
	"https://ghibliapi.herokuapp.com/films/0440483e-ca0e-4120-8c50-4c8cd9b965d6"
	],
	"species": "https://ghibliapi.herokuapp.com/species/af3910a6-429f-4c74-9ad5-dfe1c4aa04f2",
	"url": "https://ghibliapi.herokuapp.com/people/ebe40383-aad2-4208-90ab-698f00c581ab"
	}
		Each person has a field “films”, which is a list contains URIs of related films

In order to limit I/O network call, we need only have to make 2 calls:

- To fetch ALL films
- To fetch ALL people

Then we can process them in-memory to join the 2

Explanation on how to link films to people

As notice in the example, each person has a field “films”, which is a list with this format:

```
"films": [
      "https://ghibliapi.herokuapp.com/films/0440483e-ca0e-4120-8c50-4c8cd9b965d6"
]
```

Because we should have ALL data of films fetched in-memory, we do not need to hit the URI to fetch data for each film. Note that the argument of the URL is

0440483e-ca0e-4120-8c50-4c8cd9b965d6

Which is the ID of the movie. So in order to link from people to film, we do as following:

1. Build a hash table of films, in which the key is the ID of the films and content is the data of that film. Because we need to query this table frequently to match people to films, it makes sense to use a hash table as its complexity in time is O(1). Although the space complexity is O(n), we have to process the data in-memory anyway so it would not make much difference.
   For example, the hash table may look like:

```
{
      "id_of_film_harry_potter": {
          "id": "id_of_film_harry_potter",
          "title": "Harry Potter",
          "description": "Witch craft",
          "director": "J.K Rowling",
          "producer": "Hollywood",
          "release_date": "2010",
          "rt_score": "95"
      },
      "if_of_film_one_piece": {
```

```
            "id": "if_of_film_one_piece",
            "title": "One Piece",
            "description": "Kaizoku Ou ni, ore wa naru",
            "director": "Eiichiro Oda",
            "producer": "Tohei Animation",
            "release_date": "2010",
            "rt_score": "95"
        },
        "id_of_a_random_film": {
            "id": "id_of_a_random_film",
            "title": "Somthing that should not be made",
            "description": "This film should have not been made",
            "director": "Just a random guy",
            "producer": "Just a random producer",
            "release_date": "1320",
            "rt_score": "0.00001"
        }
    }
```


2.

For each person do:

Find all URLs of films related to that person (in field “films”)

For each related film, do:

*Find the data of that film in the film reference table (from step 1) **

*Insert the person data into that film. **

* These two steps benefit greatly from the fact that querying film data from reference film dataset is O(1) in time complexity.

Example of result may look like (note the field “peope_involved” is added):

```
{
       "if_of_film_one_piece": {
           "id": "if_of_film_one_piece",
           "title": "One Piece",
           "description": "Kaizoku Ou ni, ore wa naru",
           "director": "Eiichiro Oda",
           "producer": "Tohei Animation",
           "release_date": "2010",
           "rt_score": "95",
           "people_involved": {
               "id_of_monkey_dluffy": {
                   "id": "id_of_monkey_dluffy",
                   "name": "Monkey D. Luffy",
               },
               "id_of_vinsmoke_sanji": {
                   "id": "id_of_vinsmoke_sanji",
                   "name": "Vinsmoke Sanji",
               }
           }
       },
       "id_of_film_harry_potter": {
           "id": "id_of_film_harry_potter",
           "title": "Harry Potter",
           "description": "Witch craft",
           "director": "J.K Rowling",
           "producer": "Hollywood",
           "release_date": "2010",
           "rt_score": "95",
           "people_involved": {
               "id_of_harry_potter": {
                   "id": "id_of_harry_potter",
                   "name": "Harry Potter",
               }
           }
       }
}
```

Inconvenience
The downside of the approach is that the processing would be done in-memory, which may require the API to do heavy workload if Ghibli Studio API returns a huge data set.

This is where space complexity O(n) would become problematic with this approach.

Another approach should be planned if Ghibli Studio returns huge films and people dataset, as this current in-memory approach would require a vertical scaling strategy in terms of deployment and thus it is not ideal.

Test strategies

For unit tests and integration tests, as the system requires an external API to be hit, the scenario presents us a few options:

1. During the test execution, the program can hit the external API as many times it wants. However, hitting external API every time for test is considered to be a bad practice as it would:

- Increase execution time
- Increase network IO
- Make the test's behavior non-deterministic and flaky.

2. Mock the external API for the tests. However, mocking external API is considered a bad practice as well as :

- We should not mock what we do not own
- It makes our tests depend a lot on hypotheses One of the major hypotheses is that the external API does not change.
- Evolvement is difficult. Every external API change would require mirror actions on our side. Moreover, when a breaking change happens on the external API, our system breaks but the tests would still pass (as it works with mocked API).

3. Hit the external API and cache its response. These approaches satisfy the fact that we do not have to mock anything, but also extremely limit the actual external API hits during test execution. *But most importantly, we can make it FAIL when the external API introduces a breaking change. It is important to note that, when the external API changes in a non-compatible way, our tests should FAIL. It would be critically dangerous if our tests pass but the actual system fails in*

*production. An error that is explicitly thrown everywh*ere *is much better than an error passing under the ground in complete silence.*

Thus, the package `vcrpy` is used. This package records the reply of external API and replays it for future executions. You may notice that a folder named `tests\__fixtures__` is created after first execution of the tests. The next execution does not hit the external API any more, but relies on these fixtures to execute the test.

To explicitly invalidate the fixtures, just delete this folder and the next execution will create new fixtures based on actual data coming from external services.