There're 3 microservices(MS) stubs implemented for you - books, authors and frontend.

Authors and books microservices contain few pre-created records.

Frontend MS should make a calls for both books and authors MS and aggregate their responses into 1 JSON.

**Request/Response Communication**

- Authors MS should have GET endpoint api/v1/authors to return all pre-created records in JSON like:

1. [
2. {
3. "id": 1,
4. "firstName": "Loreth Anne",
5. "lastName": "White - v2"
6. },
7. ...
8. ]

- Books MS should have GET endpoint api/v1/books to return all pre-created records in JSON like:

1. [
2. {
3. "id": 1,
4. "title": "Semiosis: A Novel - v2",
5. "pages": 326,
6. "authorId": 1
7. },
8. ...
9. ]

- Frontend MS should make a call to both authors and books MS using GET endpoints created from above.

Frontend MS have GET endpoint api/v1/dashboard and return aggregated results for authors and books MS in JSON like:

1. {
2. "authors": [
3. {
4. "id": 1,
5. "firstName": "Loreth Anne",
6. "lastName": "White - v2"
7. },
8. ...
9. ],
10. "books": [
11. {
12. "id": 1,
13. "title": "Semiosis: A Novel - v2",
14. "pages": 326,
15. "authorId": 1
16. },
17. ...
18. ]
19. }

**gRPC communication:**

- Implement gRPC handlers for authors and books MS.

- Frontend MS should be able to populate aggregated data from authors and books MS via GRPC

**Message-passing communication:**

- Implement new REST PUT endpoint for **Books** MS to add a new book to the local storage (check **books/internal/service** package). To implement this task you may need to implement new struct like with followingfields

type BookAndAuthor struct {  
 ID string  
 Title string  
 Pages int  
 AuthorID int  
 FirstName string  
 LastName string  
}

- Once a new book is added new event with **BookAndAuthor** info should be posted to the RabbitMQ queue.

- **Authors** MS should consume event, that contains **BookAndAuthor** data, from the RabbitMQ and check if the author exists in the local storage (check **authors/internal/service** package). If the author is absent – it should be added to the local storage.

Useful information to start with:

* All microservices are ready to be run as Docker images, you can find **Dockerfile** for each microservice and **docker-compose.yml** to start all microservices with ports binding.
* You need to build and **package** microservice before running it in docker, check **pom.xml** file
* After microservices **.jar** files are built you can run microservices with docker-compose command.
* Useful docker-compose commands:
  1. **docker-compose up --build -** builds, (re)creates, starts, and attaches to containers for a service.
  2. **docker-compose down** -Stops containers and removes containers, networks, volumes, and images created by **up**.
* Existing **docker-compose.yml** file contains required microservices port bindings and dependencies:
  1. **Authors** MS is available by port **8094**.
  2. **Books** MS is available by port **8095**.
  3. **Frontend** MS is available by port **8096**
  4. **RabbitMQ** is available by ports **5672** and management port **15672.** You can check management console with **guest/guest** as user/password.
* Protobuf files (**.proto**) and the generated ones should be placed in a standalone Go module.
* You can use any popular framework for HTTP endpoints implementation, but the most lightweight simple, and popular at the same time is gorilla mux.

**Guides**

https://thenewstack.io/make-a-restful-json-api-go/

**Protobuf and GRPC**

[Language Guide (proto3)](https://developers.google.com/protocol-buffers/docs/proto3" \l "simple)

https://grpc.io/docs/languages/go/quickstart/

https://grpc.io/docs/languages/go/basics/

**RabbitMQ**

<https://www.rabbitmq.com/tutorials/tutorial-one-go.html>