Построение проекта big_go

Создать систему с несколькими компонентами:

- Генератор данных - создает данные о температуре, давлении и влажности с разной периодичностью имитируя асинхронность прихода.
  - Данные получают "принадлежность" к адресу и посту данного адреса.
  - Привязка ко времени - timestamp.
- Коллектор - приложение, которое получает данные через RabbitMQ
- Пользовательские приложения (User1 и User2) - получают данные от коллектора
- Инфраструктура - PostgreSQL, Redis, RabbitMQ в отдельных контейнерах

Состав проектируемой системы и как происходит работа

- **Инфраструктура коммуникаций проекта опирается на сеть big_go_network**
  - Общая сеть сервисов проекта - big_go_network создается в docker-compose.yml
- **Сервис: rabbitmq**
  - Создается как сервис (big_go_rabbitmq) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
- **Сервис: redis**
  - Создается как сервис (big_go_generator) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
  - --- Пока не используется
- **Сервис: postgresql**
  - Создается как сервис (big_go_postgres) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
  - --- Пока не используется
- **Сервис: generator**
  - Создается как сервис (big_go_generator) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
  - Подписывается на RabbitMQ как публикатор данных (producer)
  - Создает канал с rabbitmq
  - Создает очередь с именем "sensor_data"
  - Подключение к RabbitMQ происходит с использованием данных из config_rabbitmq.json

```
{
  "host": "rabbitmq",
  "port": 5672,
  "user": "guest",
  "password": "guest",
  "vhost": "/"
}
```

  - генератору назначен environment в docker-compose.yml

```
    environment:
    - RABBITMQ_HOST=rabbitmq
    - RABBITMQ_PORT=5672
    - RABBITMQ_USER=guest
    - RABBITMQ_PASSWORD=guest
```

  - Логика инициализации в /cmd/generator/main.go
- **Сервис: collector**
  - Создается как сервис (big_go_collector) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
  - Подписывается на RabbitMQ как получатель данных данных (consumer)
  - Создает канал с rabbitmq
  - В RabbitMQ использует очередь (или создает) очередь с именем "sensor_data"
  - Подключение к RabbitMQ происходит с использованием данных из config_rabbitmq.json

```
{
  "host": "rabbitmq",
  "port": 5672,
  "user": "guest",
  "password": "guest",
  "vhost": "/"
}
```

  - Логика инициализации в /cmd/collector/main.go
  - коллектору назначен environment в docker-compose.yml

```
  environment:
    - RABBITMQ_HOST=rabbitmq
    - RABBITMQ_PORT=5672
    - RABBITMQ_USER=guest
    - RABBITMQ_PASSWORD=guest
    - POSTGRES_HOST=postgres
    - POSTGRES_PORT=5432
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
    - POSTGRES_DB=big_go
    - REDIS_HOST=redis
    - REDIS_PORT=6379
```

  - Логика работы коллектора:
    - Коллектор отслеживает приход данных в цикле Обработка сообщений

```
for d := range msgs {
    ...
}
```

- Сохраняет приход в sensorData
- В каждом сообщении sensorData выявляет адресата endpoint=(user1 или user2 или ...)

```
...
var endpoint string
    if sensorData.Meta.Recipient == "User1" {
        endpoint = "http://user1:8082/data"
    } else {
        endpoint = "http://user2:8083/data"
    }
...
```

- Отправляет данному endpoint сообщение jsonData, которое извлекает из sensorData

```
...
jsonData, err := json.Marshal(sensorData)
...
resp, err := http.Post(endpoint, "application/json",
bytes.NewBuffer(jsonData))
```

- **Сервис: Приложение user1**
  - Создается как сервис (big_go_user1) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
  - Логика в /cmd/user1/main.go
    - Использует фреймворк gin (... r := gin.Default() ...)
    - Получает данные в (... var latestData []models.SensorData ...)
    - Обработчик для получения данных от коллектора

```
...
r.POST("/data", func(c *gin.Context) {
    //Тут происходит log.printf данных с метаданными в консоль
}
...
```

- Обработчик для отображения последних данных на странице

```
...
    r.GET("/", func(c *gin.Context) {
    c.HTML(http.StatusOK, "index.html", gin.H{
        "title": "User1 Dashboard",
        "data":  latestData,
```

```
    })
})
...
```

- User1 сервис запущен на порту 8082 и наблюдается как WEB страница http://localhost:8082
  - На странице отображается (User1 Dashboard) и таблица с поступающими данными
- **Сервис: Приложение user2** идентично приложению user1
  - Создается как сервис (big_go_user2) со всеми настройками в docker-compose.yml
  - Работает в сети big_go_network
  - Логика инициализации в /cmd/user2/main.go
  - User2 сервис запущен на порту 8083 и наблюдается как WEB страница http://localhost:8083
    - На странице отображается (User2 Dashboard) и таблица с поступающими данными

Структура проекта

```
big_go/
├── cmd/
│   ├── collector/
│   │   └── main.go
│   ├── generator/
│   │   └── main.go
│   ├── user1/
│   │   └── main.go
│   └── user2/
│       └── main.go
├── config/
│   ├── config.go
│   ├── opentsdb.go
│   ├── postgresql.go
│   ├── rabbitmq.go
│   └── redis.go
├── docker/
│   ├── collector/
│   │   └── Dockerfile
│   ├── generator/
│   │   └── Dockerfile
│   ├── user1/
│   │   └── Dockerfile
│   └── user2/
│       └── Dockerfile
├── internal/
│   ├── handlers/
│   ├── models/
│   ├── repository/
│   ├── routes/
│   └── services/
│       ├── generator/
│       │   └── generator.go
```

```
│       ├── collector/
│       │   └── collector.go
│       └── user/
│           └── user.go
├── docker-compose.yml
└── ...
```

Порядок запуска проекта

- **Клонирование репозитория и переход в директорию проекта:**

```
git clone <ваш-репозиторий>
cd big_go
```

- **Создание необходимых конфигурационных файлов:**
  - config_go.json - основная конфигурация приложения
  - config_postgresql.json - конфигурация PostgreSQL
  - config_redis.json - конфигурация Redis
  - config_rabbitmq.json - конфигурация RabbitMQ
- **Запуск проекта с помощью Docker Compose:**

```
docker-compose up -d
```

Эти команды запустят:

- PostgreSQL
- Redis
- RabbitMQ
- Генератор данных
- Коллектор
- User1
- User2

Проверка работы сервисов:

- **После запуска вы можете проверить работу сервисов:**
  - Веб-интерфейс RabbitMQ: http://localhost:15672 (логин: guest, пароль: guest)
  - User1 Dashboard: http://localhost:8082
  - User2 Dashboard: http://localhost:8083
- **Мониторинг логов:**

```
docker-compose logs -f
```

- **Для просмотра логов конкретного сервиса:**

```
docker-compose logs -f generator
docker-compose logs -f collector
docker-compose logs -f user1
docker-compose logs -f user2
```

- **Остановка проекта:**

```
docker-compose down
```

- **Для полной очистки (включая удаление томов):**

```
docker-compose down -v
```

Описание работы системы

- Генератор данных создает случайные данные о:
  - температуре, давлении и влажности с разной периодичностью (от 1 до 5 секунд)
  - и отправляет их в RabbitMQ.
- Коллектор
  - подписывается на сообщения из RabbitMQ,
  - обрабатывает их
  - и перенаправляет соответствующим пользовательским сервисам (User1 или User2) в зависимости от поля recipient в метаданных.
- **Пользовательские сервисы (User1 и User2) получают данные от коллектора и отображают их на веб-странице,**

Описаны

- Структура проекта с необходимыми директориями и файлами
- Dockerfile для каждого сервиса (генератор, коллектор, user1, user2)
- Docker Compose файл для оркестрации всех контейнеров
- Реализация основных компонентов:
  - Модель данных
  - Генератор данных
  - Коллектор
  - Пользовательские сервисы
  - HTML шаблон для отображения данных
- Подробный порядок запуска проекта
- Краткое описание работы системы

Для полной работоспособности проекта нужно:

- Создать все указанные файлы
- Настроить конфигурационные файлы (config_go.json, config_postgresql.json и т.д.)
- Запустить проект с помощью Docker Compose