VAKA ÇALIŞMASI - THREAD MANAGER

Hüseyin Emre İnan

hemreinan@gmail.com

19.12.2024

1. Genel Bakış

Proje Tanımı

Bu proje, kullanıcıların Sender ve Receiver thread'lerin oluşturulmasını, yönetimini ve görselleştirilmesini sağlayan bir sistemdir. Sistem, backend tarafında Spring Boot ile geliştirilmiş ve Kafka entegrasyonu kullanılarak mesajlaşma işlemleri gerçekleştirilmiştir. Frontend kısmı ise React ile geliştirilmiş olup kullanıcı dostu bir arayüz sunmaktadır.

Kullanıcılar, thread sayısını seçerek thread'lerin başlatılmasını sağlayabilir, thread'lerin durumlarını canlı olarak izleyebilir ve önceliklerini değiştirebilir. Sistem, thread'ler arasındaki veri paylaşımını yönetmek için Kafka'yı bir mesaj kuyruğu olarak kullanmaktadır.

Kullanılan Teknolojiler

- **Frontend:** ReactJS
- **Backend:** Spring Boot
- **Mesajlaşma Sistemi:** Apache Kafka
- **Dokümantasyon:** Swagger (OpenAPI)
- **Versiyon Kontrol:** Git

Proje Hedefleri

➔ **Dinamik Thread Yönetimi:**
  - ◆ Kullanıcıların, belirli sayıda Sender ve Receiver thread başlatmasını ve bu thread'lerin durumlarını yönetmesini sağlamak.

➔ **Canlı Durum İzleme:**
  - ◆ Kullanıcıların thread'lerin aktiflik durumlarını ve önceliklerini anlık olarak izleyebilmesi.

➔ **API ve Kafka Entegrasyonu:**
  - ◆ Backend ile frontend arasındaki iletişimi sağlayacak API'ler ve Kafka mesajlaşma altyapısı.

➔ **Kullanıcı Dostu Arayüz:**
  - ◆ Kullanıcı dostu thread oluşturma, kontrolü.Thread durumlarının net ve kolay anlaşılır şekilde görselleştirilmesi.

2. Geliştirme Ortamı ve Kurulum

Sistem Gereksinimleri

Aşağıda, projeyi çalıştırabilmek için gerekli olan sistem gereksinimleri ve araçlar listelenmiştir:

- ➔ **İşletim Sistemi:**
  - ◆ Windows, Linux veya macOS
- ➔ **Java:**
  - ◆ Java 17 + (Spring Boot için)
- ➔ **Node.js:**
  - ◆ Node.js 16+ (React uygulaması için)
- ➔ **Kafka:**
  - ◆ Apache Kafka 2.8.0 veya daha yüksek bir versiyon (mesajlaşma için)
- ➔ **IDE/Metin Editörü:**
  - ◆ IntelliJ IDEA, VS Code veya benzeri

Backend Kurulumu

1. Gerekli Bağımlılıkların Yüklenmesi:

- Backend projesi Spring Boot ile geliştirilmiştir.
- Proje, Maven kullanılarak yönetilmektedir.
- Maven bağımlılıkları yüklemek için aşağıdaki komutu çalıştırılır:

```
mvn clean install
```

- Kafka bağımlılıkları da `pom.xml` dosyasına eklenir.

```
<dependency>

    <groupId>org.springframework.kafka</groupId>

    <artifactId>spring-kafka</artifactId>

    <version>2.8.0</version>

</dependency>
```

2. **Kafka'nın Çalıştırılması:**

   Kafka’nın çalışabilmesi için bir Kafka Broker' ına ihtiyaç vardır. Kafka' yı çalıştırmak için aşağıdaki komutlar kullanılabilir:

```
# Kafka'yı çalıştırmak için ZooKeeper başlatılmalıdır

bin/zookeeper-server-start.sh config/zookeeper.properties

bin/kafka-server-start.sh config/server.properties
```

3. **Backend Uygulamasını Başlatma:**

   Spring Boot uygulamasını başlatmak için aşağıdaki komut çalıştırılabilir ya da ide üzerinden ‘SpringBootApplication.java’ isimli spring boot application dosyası run edilebilir:

```
mvn spring-boot:run
```

4. **Backend Server Portları**

   Uygulama ayağa kalktıktan sonra application.properties dosyasındaki ayarlara göre

   - Receiver için: 8081
   - Sender için : 8082

   portlarında çalışmaya başlar.

Frontend Kurulumu

- Bu projede React uygulaması, hızlı geliştirme süreçleri için **Vite** kullanılarak yapılandırılmıştır.
- Vite, React gibi modern JavaScript framework' leri ile çalışan hızlı bir geliştirme sunucusu ve derleyicidir.

Vite Kullanarak Kurulum Adımları

1. **Gerekli Bağımlılıkların Yüklenmesi:**

   Vite'i kurmak için, terminal veya komut satırında aşağıdaki komutu kullanarak yeni bir Vite projesi oluşturulur, Bu komut, `my-project` adlı bir dizin oluşturacak ve React şablonunu içeren bir proje başlatacaktır :

   ```
   npm create vite@latest my-project --template react
   ```

2. **Proje Dizinine Giriş:**

   Proje dizinine giriş yapılır ve ardından bağımlılıkları yüklemek için aşağıdaki komut çalıştırılır:

   ```
   cd my-project

   npm install
   ```

3. **Vite Uygulamasını Başlatma:**

   Projeyi başlatmak için aşağıdaki komut çalıştırılır, Bu komut, uygulamyı yerel geliştirme sunucusunda başlatır ve genellikle `http://localhost:5173` adresinde erişilebilir hale gelir:

   ```
   npm run dev
   ```

3. Proje Mimarisi

Genel Mimarisi

Proje, iki ana katmandan oluşan bir mikro hizmet mimarisi kullanılarak tasarlanmıştır: **Frontend (React)** ve **Backend (Spring Boot)**. Bu yapı, kullanıcı arayüzü ile backend sunucusu arasında temiz bir ayrım sağlar ve her iki tarafın bağımsız olarak geliştirilmesine olanak tanır. Ayrıca, **Apache Kafka** kullanılarak veri iletimi ve asenkron işleme sağlanmaktadır.

- ➔ **Frontend (React + Vite)**:
  - ◆ Kullanıcı arayüzü, React ile geliştirilmiştir ve Vite kullanılarak optimize edilmiştir. Frontend, kullanıcı etkileşimlerini yönetir ve backend ile REST API aracılığıyla iletişim kurar.
- ➔ **Backend (Spring Boot)**:
  - ◆ Spring Boot, uygulamanın backend kısmını yönetir. Thread yönetimi, Kafka ile mesajlaşma ve API servislerinin yönetimi burada yapılır.
- ➔ **Kafka**:
  - ◆ Kafka, veri iletimini yöneten bir mesajlaşma sistemi olarak, Sender ve Receiver thread' lerinin veri paylaşımını ve işleme sırasını kontrol eder. Kafka, asenkron veri akışını sağlar ve thread' ler arasında veri iletiminde kullanılacak kuyruğu yönetir.

Mimari Katmanlar

1. **Frontend Katmanı (React)**
   - Kullanıcıların thread sayısını seçebileceği ve bu seçime göre backend' in thread başlatacağı bir UI sağlar.
   - Kullanıcılar, thread'lerin durumlarını (aktif/pasif) değiştirebilir, önceliklerini ayarlayabilir ve thread'leri başlatıp durdurabilir.
   - React bileşenleri, **Vite** ile geliştirilmiş olup hızlı geliştirme ve hot reload özelliklerine sahiptir.
   - Frontend, backend API' leriyle HTTP istekleri yaparak thread' lerin durumlarını yönetir.

2. **Backend Katmanı (Spring Boot)**
   - **Spring Boot** : mikroservis mimarisiyle geliştirilmiştir.
   - **Kafka Entegrasyonu**: Backend, Kafka'ya entegre edilmiştir ve thread'ler arasındaki veri iletimi Kafka üzerinden yapılır. Sender thread'leri veri gönderirken, Receiver thread'leri bu veriyi alır ve işler.

- **Thread Yönetimi**: Spring Boot, backend'de thread'leri yönetir. Thread'lerin durumu, önceliği ve yaşam döngüsü Spring Boot uygulaması tarafından izlenir.
- **REST API**: Frontend ile backend arasındaki iletişim, REST API' leri aracılığıyla yapılır. Kullanıcılar, thread sayısını seçebilir, thread'lerin durumunu güncelleyebilir ve önceliklerini değiştirebilir.
- **Swagger/OpenAPI**: API dokümantasyonu için Swagger kullanılmaktadır. Kullanıcılar ve geliştiriciler API'lerin işlevselliğini ve kullanımını Swagger arayüzü üzerinden görüntüleyebilir.

3. **Kafka Katmanı**
   - **Mesajlaşma Sistemi**: Kafka, asenkron veri iletimi için kullanılır. Backend, thread' ler arasında veri paylaşımını Kafka üzerinden yönetir.
   - **Producer ve Consumer**: Backend' deki sender thread'leri, Kafka'nın producer' ları olarak veri üretir. Receiver thread' leri ise Kafka'nın consumer'ları olarak bu veriyi tüketir.
   - **Topic Yapıları**: Kafka topic' leri, thread' ler arası veri iletişimini organize eder. Sender thread' leri belirli bir topic' e veri gönderirken, Receiver thread' leri aynı topic' ten veri alır.
   - **Kafka Yöneticisi**: Kafka'nın durumunu izlemek ve yönetmek için bir yönetici arayüzü kullanılır.

Katmanlar Arası İletişim

- **Frontend ve Backend İletişimi (REST API)**
  - Frontend, backend API'leri aracılığıyla thread başlatma, durumu güncelleme, öncelikleri değiştirme gibi işlemleri yapar.
  - API'ler JSON formatında veri alır ve geri döner.
  - API'lerin tümü Swagger dokümantasyonu ile kullanıcıya sunulmaktadır.

Örnek API çağrıları:

➢ Belirli sayıda thread başlatır :
- **POST** `/api/receiver/start-thread`

➢ Tüm thread'lerin durumunu getirir.
- **GET** `/api/senders/get-thread-infos`

➢ Bir thread'in önceliğini değiştirir.
- **PUT** `/api/threads/change-thread-priority`

- **Backend ve Kafka İletişimi**
    - Backend, Kafka kullanarak asenkron veri iletimi yapar.
    - Sender thread'leri Kafka producer olarak veriyi üretirken, Receiver thread'leri Kafka consumer olarak bu veriyi tüketir.
    - Kafka, thread'lerin veri paylaşımını yönetir ve asenkron işleme sağlar.

- **Frontend ve Kafka İletişimi**
    - Frontend doğrudan Kafka ile iletişim kurmaz; bunun yerine backend API'leri üzerinden Kafka ile etkileşimde bulunur.
    - Backend, frontend'ten gelen istekleri alır ve Kafka ile iletişim kurarak veri akışını yönetir.

4. Kod Yapısı ve Organizasyonu

Proje Yapısı

Proje, hem frontend (React) hem de backend (Spring Boot) bileşenlerinden oluşan iki ana modüle ayrılmıştır. Her modülün kendi bağımsız dizin yapıları ve organizasyonları vardır, böylece proje daha modüler ve sürdürülebilir hale gelir.

- ### Frontend Yapısı

  React uygulaması **Vite** ile geliştirilmiş olup, hızlı geliştirme ve optimizasyon özelliklerinden faydalanmaktadır. Aşağıda, frontend kısmının temel dizin yapısı yer almaktadır:

  - ➔ **/components**:
    - ◆ Uygulamanın temel bileşenleri burada yer alır. Her bileşen, kendi CSS dosyasıyla birlikte organize edilmiştir. Örneğin, `AddThread.jsx` bileşeni, `AddThread.css` dosyasına sahip olabilir.
  - ➔ **/services**:
    - ◆ Backend ile iletişim sağlayan API servislerini içerir. Bu servisler, backend API'lerine HTTP istekleri gönderir.
  - ➔ **/contexts**:
    - ◆ React Context API ile global state yönetimi yapılır. Thread'lerin durumu ve önceliği gibi bilgiler burada yönetilebilir.
  - ➔ **/styles**:
    - ◆ Uygulamanın stil dosyaları yer alır. Global stiller ve bileşen bazlı stiller burada organize edilir.
  - ➔ **/utils**:
    - ◆ Yardımcı fonksiyonlar (örneğin, API isteklerini işlemek veya özel hesaplamalar yapmak) burada bulunur.

- ## Backend Yapısı

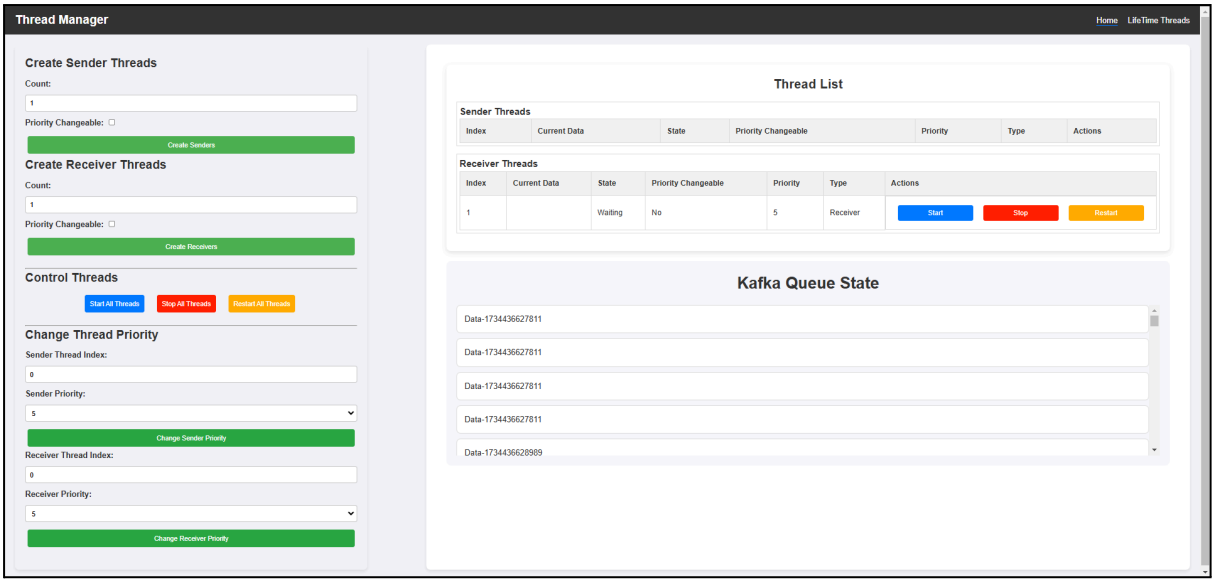
Spring Boot ile yazılan backend kısmı, mikro hizmetler mimarisi kullanarak modüler şekilde yapılandırılmıştır. Kafka ile entegrasyon, thread yönetimi ve API servisleri burada yer alır. Aşağıda backend kısmının temel dizin yapısı yer almaktadır:

- ➔ **/controller**:
  - ◆ REST API endpoint'leri burada yer alır. Thread başlatma, durumu güncelleme ve öncelik değiştirme gibi işlemler bu katmanda yapılır.
- ➔ **/model**:
  - ◆ Uygulamanın veri modellerini içerir. Kafka mesajları ve thread yönetimi ile ilgili sınıflar burada tanımlanır.
- ➔ **/service**:
  - ◆ Uygulamanın iş mantığı burada yer alır. Kafka ile veri iletimi, thread'lerin yönetimi ve veri işleme gibi işlemler bu katmanda yapılır.
- ➔ **/config**:
  - ◆ Kafka konfigürasyonları, uygulama ayarları ve diğer yapılandırmalar burada bulunur.
- ➔ **/test**:
  - ◆ Backend için yazılan birim ve entegrasyon testlerinin bulunduğu dizindir. API testleri, Kafka servis testleri bu katmanda yapılır.

5. Kullanıcı Arayüzü ve Kullanım Talimatları(UI/UX)

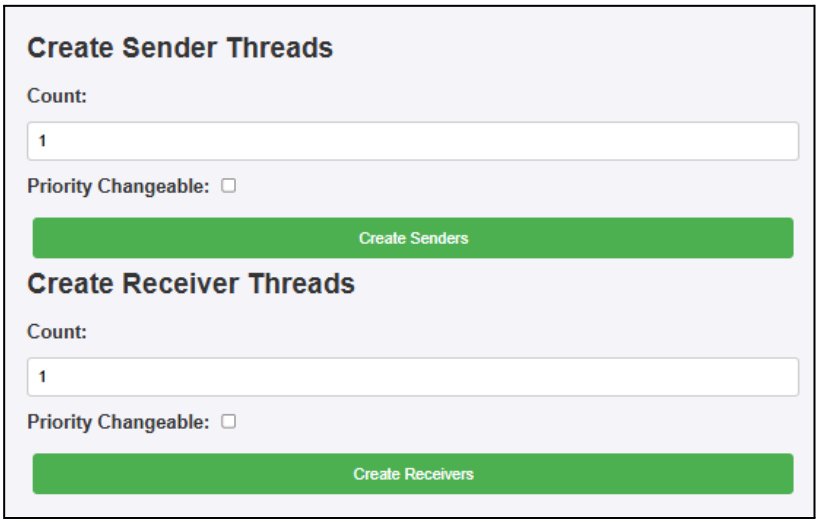
UI/UX Tasarımı

Proje, kullanıcıların thread'lerin yönetimini basit ve görsel olarak anlaşılır bir şekilde yapabilmesini sağlamak amacıyla tasarlanmıştır. UI, kullanıcı dostu bir deneyim sunacak şekilde düşünülmüş ve threadleri yönetmek ve listelemek için kullanılan her bir bileşenin görsel olarak belirgin bir şekilde ayrılması sağlanmıştır. Ayrıca, sistemin threadlerin gerçek zamanlı durum, index, priority gibi bilgilerini gösterebilmesi amacıyla canlı güncellemeler alan tablolar kullanılmıştır.

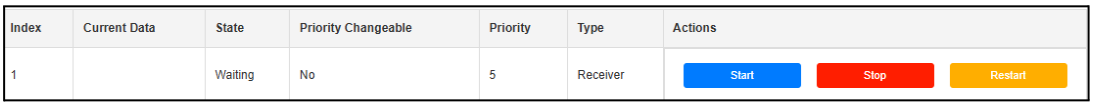


Arayüz Bileşenleri ve Talimatlar

1. **Thread Seçimleri ve Yönetimi**
    - Thread üzerinde ‘Start’ - ’Stop’ - ’Restart’ - ’Change Priority’ işlemleri yapılabilir.
    - Kullanıcı, Sender Thread ve Receiver Thread olmak üzere istediği türde oluşturmak istediği thread sayısını seçebilir. Seçilen bu sayıya göre backend, gerekli sayıda sender ve receiver thread 'lerini oluşturur.

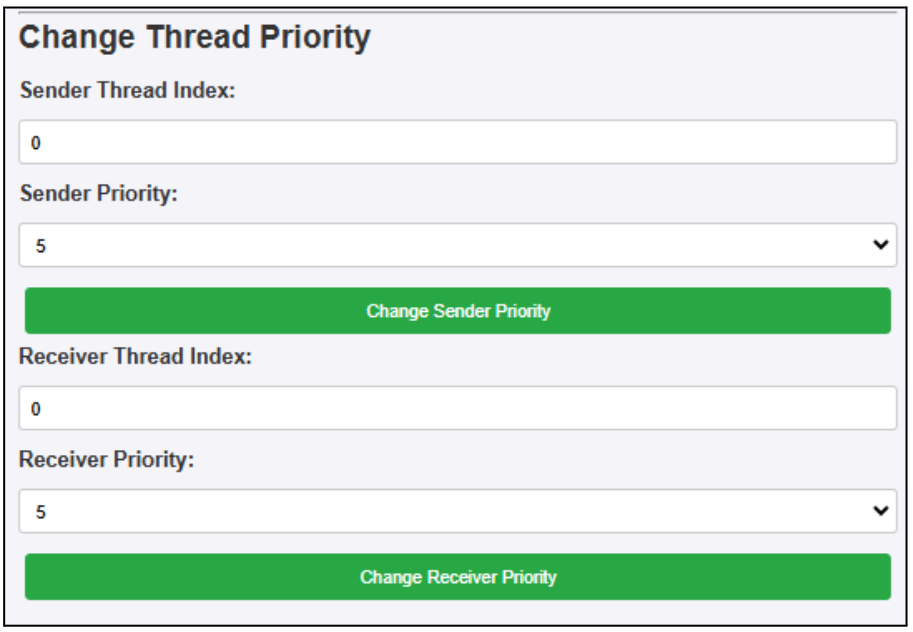


- Thread oluşturma aşamasında, ilgili thread e ait öncelik durumunun değişebilir olup olmadığı bilgisinin de seçilmesi gerekmektedir.
- Kullanıcı, threadlerin dinamik olarak bilgilerinin yer aldığı tablodan başlatmak ya da durdurmak istediği threadi seçebilir ve seçilen thread backend tarafından isteğe göre başlatılır ya da durdurulur.

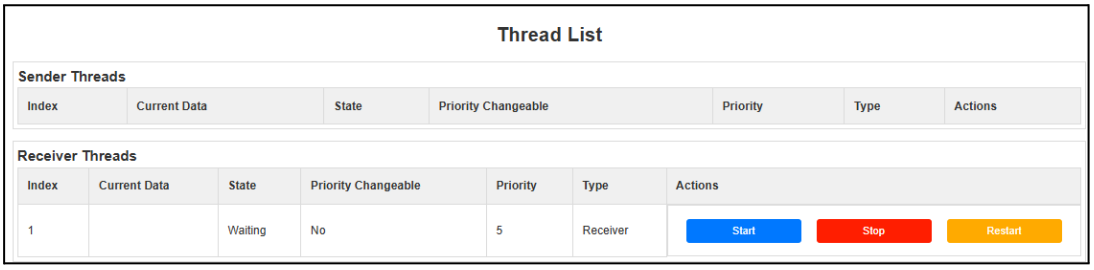


- ‘’Waiting’ durumundaki thread ‘start’ denilerek başlatılabilir
- ‘Stopped’ durumundaki thread, ‘start’ işlemi ile başlatılamaz, yeniden başlatılması için ‘’restart’ edilmesi gerekir.
- Bir thread in durdurularak ‘Stopped’ durumuna geçebilmesi için durumu ‘Running’ olmalıdır.
- Bir thread in restart denilerek başlatılması için durumu ‘Stopped’ olmalıdır.
- Kullanıcı, durdurduğu bir thread i tablodan seçerek yeniden başlatabilir.
- Kullanıcı, tüm waiting durumundaki threadleri aynı anda başlatabilir.
- Kullanıcı, tüm running durumundaki threadleri aynı anda durdurabilir.
- Kullanıcı tüm stopped durumdaki threadleri aynı anda yeniden başlatabilir

- Kullanıcı, istediği türde bir thread in önceliğini 1 ile 10 arasında güncelleyebilir. Önceliği güncellenmek istenen thread in durumu güncellenebilir olmalıdır.

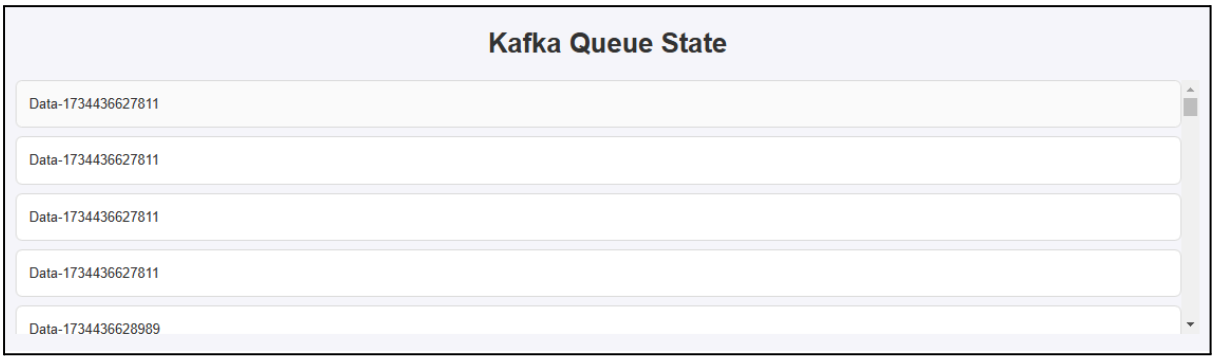


2. **Thread Durumu ve Gözetimi**
    - Kullanıcı, tüm thread' lerin index, data, durum, öncelik, tür bilgilerini ekrandan dinamik olarak izleyebilir. Thread'lerin durumu, gerçek zamanlı olarak backende takip edilir ve frontend tarafında güncellenir. Kullanıcı bir thread'in durumu üzerinde değişiklik yaparsa (başlatma, durdurma veya öncelik değişikliği), bu değişiklik anlık olarak tabloda belirtilir.



3. **Queue Durumu Görselleştirme ve Gözetimi**

    - Kullanıcı, sender ve receiver thread'lerinin veri paylaşımında kullanılan **queue'nun** anlık durumunu görsel olarak izleyebilir. Queue, veri eklendikçe dinamik olarak uzayan bir unordered list ile görselleştirilmiştir. Queue'nun içerdiği veri, kullanıcıya net bir şekilde sunulur.



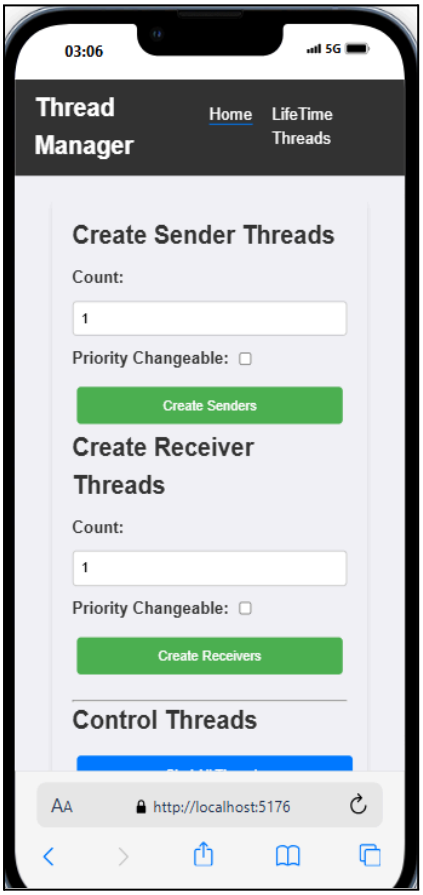
Kullanıcı Etkileşimi

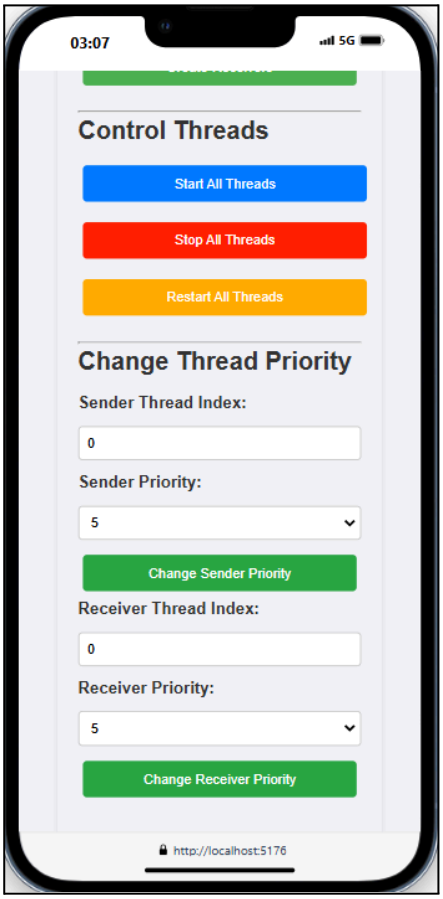
- **Real-time Feedback**:
    - Kullanıcı, thread sayısını seçtikten sonra backend'e istek gönderilir ve bu işlem anlık olarak kullanıcıya geri bildirilir. Backend'ten gelen veriler frontend'de canlı olarak güncellenir. Örneğin, thread'lerin durumu anlık olarak gösterilir.
- **Dinamik Güncellemeler**:
    - UI, kullanıcıların yaptığı değişikliklere hemen tepki verir. Thread başlatma, durdurma veya öncelik değişikliği gibi işlemler sonucunda sayfa yeniden yüklenmeden anında UI üzerinde yansır.
- **Hata ve Başarı Mesajları**:

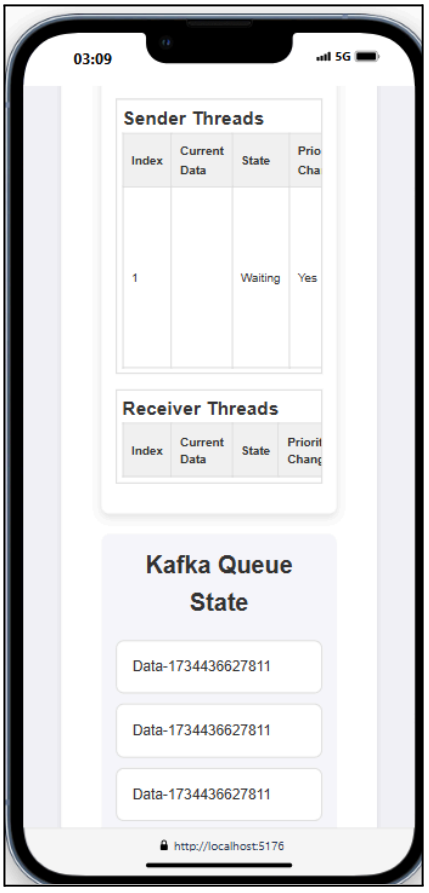
- Kullanıcı, işlemleri sırasında herhangi bir hata ile karşılaşırsa, ekranda uygun hata mesajları gösterilir. Başarılı işlemler de aynı şekilde başarı mesajlarıyla bildirilir.

Mobil Uyumluluk

Uygulama, mobil uyumlu olacak şekilde responsive tasarlanmıştır. Farklı cihazlarda ve ekran boyutlarında rahat kullanım sağlanır. Örneğin, mobil cihazlarda thread seçim kutuları ve butonlar daha büyük boyutlarla yerleştirilmiş, ekran boyutuna göre uygun şekilde hizalanmıştır.







6. API Tasarımı ve Entegrasyonu

API Genel Yapısı

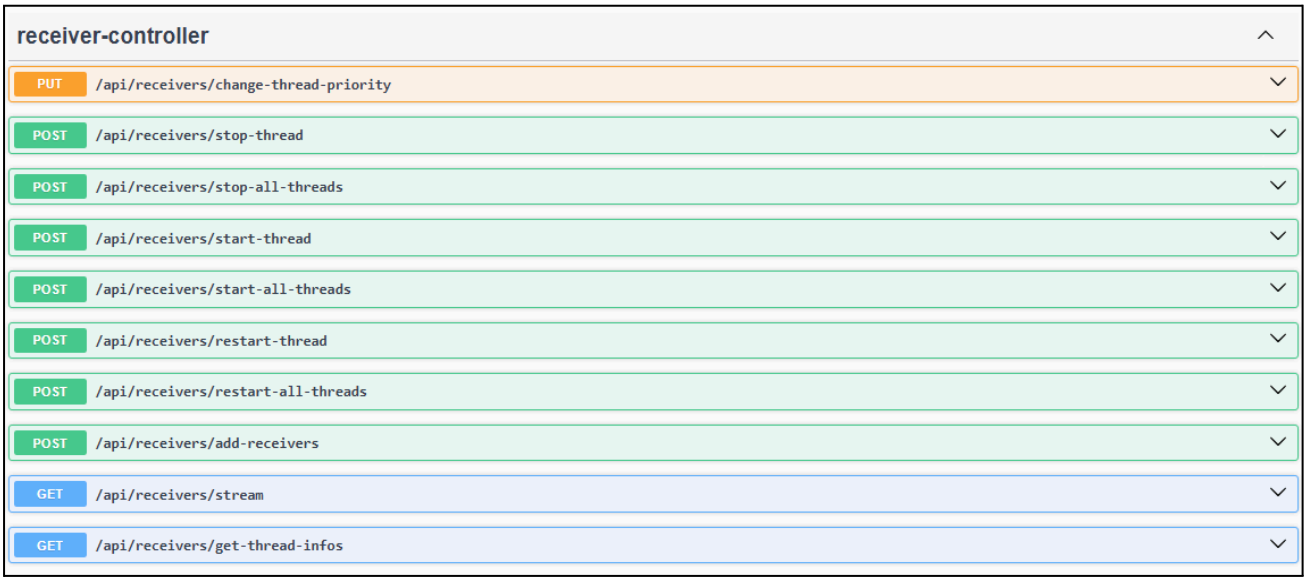
Bu projede, frontend (React) ile backend (Spring Boot) arasındaki iletişim, RESTful API'ler ve **Server-Sent Events (SSE)** aracılığıyla sağlanmaktadır. API'ler, sender ve receiver thread'lerinin yönetimini ve Kafka queue durumunun izlenmesini mümkün kılar. Frontend, bu API'leri kullanarak backend ile veri alışverişi yapar ve kullanıcı arayüzünde anlık güncellemeleri takip eder.

API Endpoint'leri

Aşağıda, backend tarafında sunulan API endpoint'lerinin detayları ve işlevleri verilmiştir. Bu API'ler sender, receiver thread'lerinin yönetimi ve Kafka queue durumu ile ilgili işlemleri sağlar.

Receiver API (ReceiverController)

Aşağıda Swagger UI ekran görüntüsü üzerinden receiver endpointlerini görebilirsiniz



1. **Receiver Thread Oluşturma**
   - ➔ **Endpoint**: `POST /api/receivers/add-receivers`
   - ➔ **Açıklama**: Belirli sayıda receiver thread'inin oluşturulmasını sağlar.
   - ➔ **Request Parametreleri**:
     - ◆ `count`: Oluşturulacak receiver thread'lerinin sayısı.
     - ◆ `isPriorityChangeable`: Thread'lerin önceliğinin değiştirilebilir olup olmadığı.
   - ➔ **Response:**
     - ◆ {"message": "5 receiver threads created."}

2. **Receiver Thread Başlatma**
   - ➔ **Endpoint**: `POST /api/receivers/start-thread`
   - ➔ **Açıklama**: Belirli bir receiver thread'ini başlatır.
   - ➔ **Request Parametreleri**:
     - ◆ `index`: Başlatılacak thread'in index numarası.
   - ➔ **Response:** {
     - ◆ "message": "1. Receiver thread started."}

3. **Tüm Receiver Thread'lerini Başlatma**
   - ➔ **Endpoint:** `POST /api/receivers/start-all-threads`
   - ➔ **Açıklama:** Tüm bekleyen receiver thread'lerini başlatır.
   - ➔ **Response:**
     - { "message": "All Waiting Receiver Threads started."}

4. **Receiver Thread Durdurma**
   - ➔ **Endpoint:** `POST /api/receivers/stop-thread`
   - ➔ **Açıklama:** Belirli bir receiver thread'ini durdurur.
   - ➔ **Request Parametreleri:**
     - ◆ `index`: Durdurulacak thread'in index numarası.
   - ➔ **Response:**
     - { "message": "Receiver thread stopped."}

5. **Tüm Receiver Thread'lerini Durdurma**
   - ➔ **Endpoint**: `POST /api/receivers/stop-all-threads`
   - ➔ **Açıklama**: Tüm aktif receiver thread'lerini durdurur.
   - ➔ **Response**
     - { "message": "All Running Receiver Threads stopped."}

6. **Receiver Thread Yeniden Başlatma**
   - ➔ **Endpoint**: `POST /api/receivers/restart-thread`
   - ➔ **Açıklama**: Belirli bir receiver thread'ini yeniden başlatır.
   - ➔ **Request Parametreleri**:
     - ◆ `index`: Yeniden başlatılacak thread'in index numarası.
   - ➔ **Response**
     - { "message": "Receiver thread restarted."}

7. **Tüm Receiver Thread'lerini Yeniden Başlatma**
   - ➔ **Endpoint**: `POST /api/receivers/restart-all-threads`
   - ➔ **Açıklama**: Tüm durdurulmuş receiver thread'lerini yeniden başlatır.
   - ➔ **Response**:
     - { "message": "All Stopped Receiver threads restarted."}

8. **Receiver Thread Önceliğini Değiştirme**
   - ➔ **Endpoint**: `PUT /api/receivers/change-thread-priority`
   - ➔ **Açıklama**: Belirli bir receiver thread'inin önceliğini değiştirir.
   - ➔ **Request Parametreleri**:
     - ◆ `index`: Thread'in index numarası.
     - ◆ `priority`: Yeni öncelik değeri.
   - ➔ **Response**
     - { "message": "Receiver thread priority set to 1."}

9. **Receiver Thread Bilgilerini Alma**
   - ➔ **Endpoint**: `GET /api/receivers/get-thread-infos`
   - ➔ **Açıklama**: Tüm receiver thread'lerinin durum bilgilerini alır.
   - ➔ **Response**:
     - [{

       "threadId": 1, "status": "active","priority": 1

       },

       {

       "threadId": 2, "status": "inactive", "priority": 2

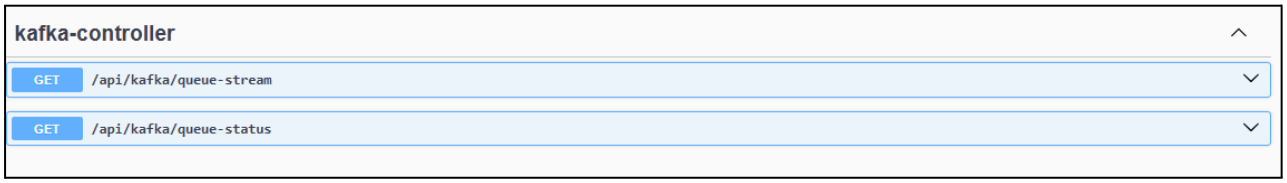
       }

       ]

10. Receiver Thread Güncellemelerini İzleme (SSE)

- ➔ **Endpoint:** `GET /api/receivers/stream`
- ➔ **Açıklama**: Receiver thread'lerinin güncellemelerini real-time olarak takip etmek için SSE kullanır.
- ➔ **Response**: Bu endpoint, sürekli veri akışı sağlayan bir SSE emitter döner.

Kafka API (KafkaController)



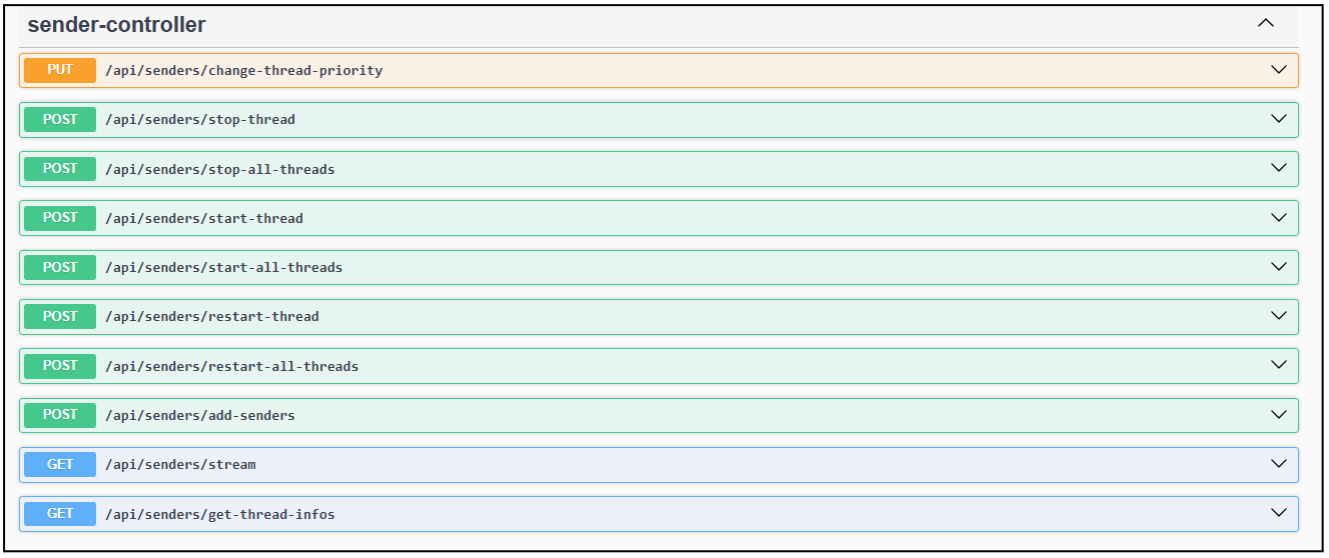
1. Queue Durumunu Almak

- ➔ **Endpoint:** `GET /api/kafka/queue-status`
- ➔ **Açıklama:** Kafka queue'sunun mevcut durumunu alır.
- ➔ **Response:**
  - { "messages": ["message1", "message2", "message3"] }

2. Queue Güncellemelerini İzlemek (SSE)

- ➔ **Endpoint:** `GET /api/kafka/queue-stream`
- ➔ **Açıklama:** Kafka queue'sunun güncellemelerini real-time olarak izler.
- ➔ **Response:** Bu endpoint, sürekli veri akışı sağlayan bir SSE emitter döner

Sender API (SenderController)



1. **Sender Thread Oluşturma**
   - ➔ **Endpoint:** `POST /api/senders/add-senders`
   - ➔ **Açıklama:** Belirli sayıda sender thread'inin oluşturulmasını sağlar.
   - ➔ **Request Parametreleri:**
     - ◆ `count`: Oluşturulacak sender thread'lerinin sayısı.
     - ◆ `isPriorityChangeable`: Thread'lerin önceliğinin değiştirilebilir olup olmadığı.
   - ➔ **Response:**
     - { "message": "5 sender threads created."}

2. **Sender Thread Başlatma**
   - ➔ **Endpoint:** `POST /api/senders/start-thread`
   - ➔ **Açıklama:** Belirli bir sender thread'ini başlatır.
   - ➔ **Request Parametreleri:**
     - ◆ `index`: Başlatılacak thread'in index numarası.
   - ➔ **Response:**
     - { "message": "1. Sender thread started."}

3. **Tüm Sender Thread'lerini Başlatma**
   - ➔ **Endpoint:** `POST /api/senders/start-all-threads`
   - ➔ **Açıklama:** Tüm bekleyen sender thread'lerini başlatır.
   - ➔ **Response:**
     - { "message": "All Waiting Sender Threads started."}

4. **Sender Thread Durdurma**
   - ➔ **Endpoint:** `POST /api/senders/stop-thread`
   - ➔ **Açıklama: Belirli bir sender thread'ini durdurur.**
   - ➔ **Request Parametreleri:**
     - ◆ **`index`:** Durdurulacak thread'in index numarası.
   - ➔ **Response:**
     - { "message": "Sender thread stopped."}

5. **Tüm Sender Thread'lerini Durdurma**
   - ➔ **Endpoint:** `POST /api/senders/stop-all-threads`
   - ➔ **Açıklama:** Tüm aktif sender thread'lerini durdurur.
   - ➔ **Response:**
     - { "message": "All Running Sender Threads stopped."}

6. **Sender Thread Yeniden Başlatma**
   - ➔ **Endpoint**: `POST /api/senders/restart-thread`
   - ➔ **Açıklama**: Belirli bir sender thread'ini yeniden başlatır.
   - ➔ **Request Parametreleri**:
     - ◆ `index`: Yeniden başlatılacak thread'in index numarası.
   - ➔ **Response:**
     - { "message": "Sender thread restarted."}

7. **Tüm Sender Thread'lerini Yeniden Başlatma**
   - ➔ **Endpoint**: `POST /api/senders/restart-all-threads`
   - ➔ **Açıklama**: Tüm durdurulmuş sender thread'lerini yeniden başlatır.
   - ➔ **Response:**
     - {"message": "All Stopped Sender threads restarted."}

8. **Sender Thread Önceliğini Değiştirme**
   - ➔ **Endpoint:** `PUT /api/senders/change-thread-priority`
   - ➔ **Açıklama:** Belirli bir sender thread'inin önceliğini değiştirir.
   - ➔ **Request Parametreleri:**
     - ◆ `index`: Thread'in index numarası.
     - ◆ `priority`: Yeni öncelik değeri**.**
   - ➔ **Response:**
     - { "message": "Sender thread priority set to 1."}

9. **Sender Thread Bilgilerini Alma**
   - ➔ **Endpoint:** `GET /api/senders/get-thread-infos`
   - ➔ **Açıklama:** Tüm sender thread'lerinin durum bilgilerini alır.
   - ➔ **Response:**
     - [{ "index": 0, "currentData": null, "priorityChangeable": true,

       "type": "SENDER", "priority": 6, "threadState": "WAITING"

       }]

10. **Sender Thread Güncellemelerini İzleme (SSE)**
    - ➔ **Endpoint:** `GET /api/senders/stream`
    - ➔ **Açıklama:** Sender thread'lerinin güncellemelerini real-time olarak takip etmek için SSE kullanır.
    - ➔ **Response:** Bu endpoint, sürekli veri akışı sağlayan bir SSE emitter döner.

Frontend Entegrasyonu

Frontend tarafında, API'lerle entegrasyon için Axios kütüphanesi kullanılmıştır. Axios, HTTP istekleri göndermek ve yanıtları almak için tercih edilen bir araçtır. API'lerle yapılan işlemler aşağıdaki gibi listelenebilir:

1. **Thread Oluşturma:**
    - Frontend, kullanıcı tarafından belirlenen thread sayısını backend'e gönderir ve threadler oluşturulur. API'den gelen yanıtla kullanıcıya bildirim yapılır.
2. **Thread Durumu Güncelleme:**
    - Kullanıcı, bir thread'in durumunu değiştirdiğinde bu değişiklik backend'e gönderilir.
3. **Queue Durumu Almak:**
    - Backend'ten queue'nun mevcut durumunu alarak frontend'de gösterir.
4. **Thread Güncellemelerini Anlık İzleme (SSE)**
    - Server-Sent Events (SSE) kullanarak frontend, backend'den gelen thread güncellemelerini anlık olarak izler. Bu sayede, kullanıcıların thread durumlarıyla ilgili real-time (gerçek zamanlı) bilgilere erişmeleri sağlanır.
5. **Kafka Durumu ve Güncellemeleri İzleme**
    - Frontend, Kafka'nın queue durumunu izlemek için `GET /api/kafka/queue-stream` endpoint'ine abone olur. Bu, Kafka queue'sundaki değişiklikleri gerçek zamanlı olarak gösterir.