15/04/2021

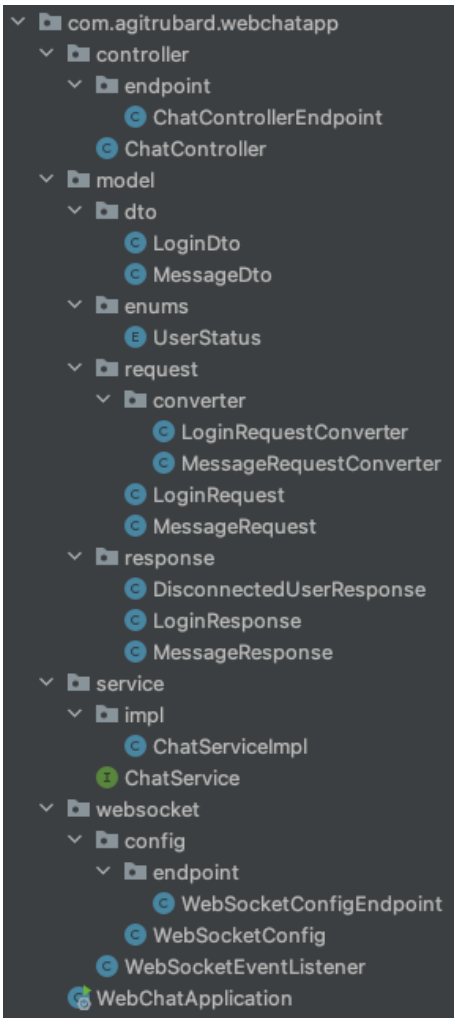
Agit Rubar DEMİR
180205011012

İLERLEME RAPORU 1

Geliştirme Sürecinde Yapılanlar;

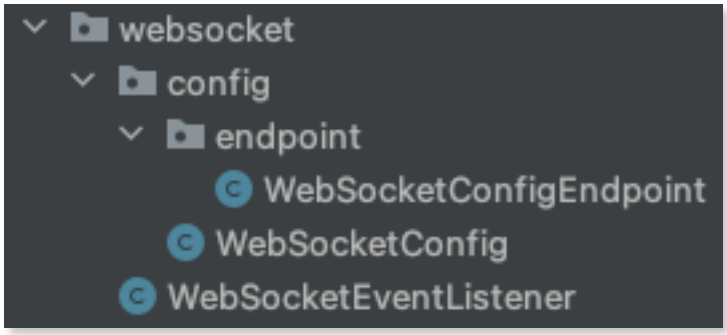
- Hangi teknolojilerin kullanılacağı belirlendi.
- Proje tanım dökümanı hazırlandı, GitHub'a README dosyası olarak eklendi.
- Kullanılacak paket yapısı, Back-End mimarisi belirlendi.
- Back-End geliştirmeleri tamamlandı.

Bu süreçte commit'ler atıldı, GitHub'a push edildi.



Paket Yapısı

Back-End



WebSocket

Config, Endpoint

websocket

↘

WebSocketEventListener

WebSocket üzerinde yapılan işlemleri dinlememizi sağlayan sınıfımız. Örneğin bağlanma ve çıkış takibini buradan yapabiliyoruz. Burada herhangi bir log mesajı ekleyebiliyoruz.

websocket

↘

config

↘

WebSocketConfig

WebSocket konfigürasyonlarının yapıldığı sınıftır. Bu sınıf ***WebSocketMessageBrokerConfigurer*** interface'inden implement edildi. Burada registry'ler kullanıldı, bu registry'ler Front-End ile bağlantı kurmamızı sağlayan hangi Endpoint'i dinleyeceğimizi belirlediğimiz classlar.

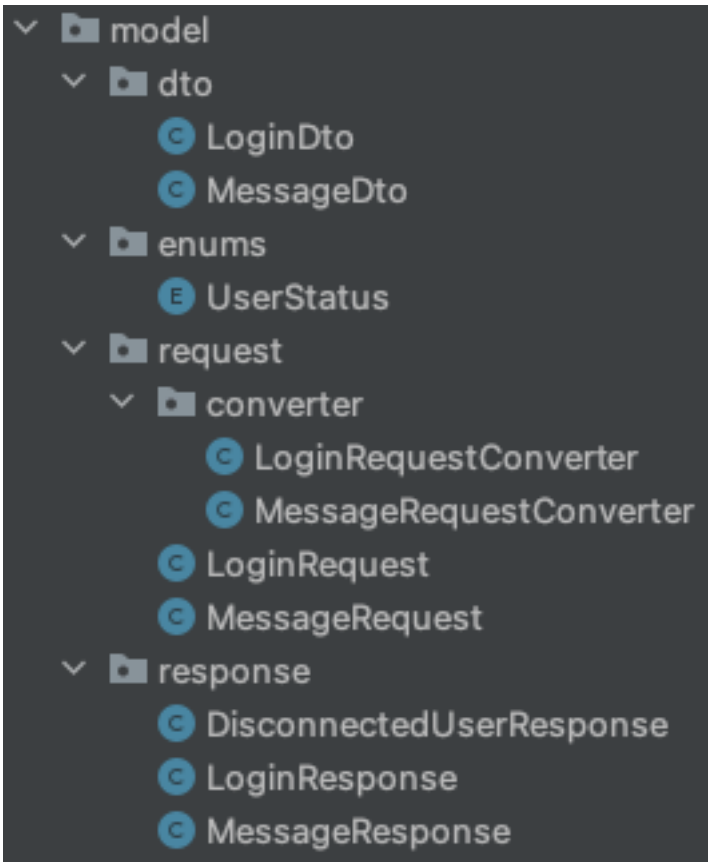
websocket

config

endpoint

WebSocketConfigEndpoint

WebSocketConfig sınıfında kullanılan Endpointlerin kontrol edilebilirliğini sağlamak ve HardCore dediğimiz sabit kodlamadan kaçınmak için oluşturulan ve static final değerlerin tutulduğu bir sınıftır.



Model

DTO, Enums, Request, Response

model

↘

dto

↘

MessageDto

MessageDto, gönderilen mesajın içerisinde bulunması gereken alanları belirlediğimiz bir Message Model sınıfı diyebiliriz.

model

↳

dto

↳

LoginDto

LoginDto, bir kullanıcı giriş yaptığında bu istek içerisinde bulunması gereken alanları belirlediğimiz bir Login Model sınıfı diyebiliriz.

Login ve Message Dto'ların farklı olmasının sebebi aslında içerisinde farklı değerler barındırması. Örneğin Login yaparken bizim mesajımızın içeriğini tutacak ihtiyacımız olmadığı için, sadece login yaptığımızda gerekli bilgileri tutacak bir sınıfa ihtiyacımız oluyor. Bunun dışında geliştirme yaparken, diyelim ki Login yaparken ekstra değişkenlere ihtiyacımız oldu, MessageDto'ya karışmadan sadece LoginDto içerisine eklemeleri yapmamız bunu kullanmamıza yeterli oluyor bu da geliştirme sırasında kodumuzun geliştirmelere uyumluluğunu arttırmamıza yardımcı oluyor.

model

↳

enums

↳

UserStatus

UserStatus, adından da anlaşılacağı üzere, kullanıcıların Chat'te olup olmadığını, chat'e katıldığında veya çıktığında yapılacak işlemleri ona göre şekillendirebileceğimiz enum değerler bulunuyor.

model

request

LoginRequest

LoginRequest sınıfı içerisinde bir kullanıcı giriş yaptığında alınması gereken parametreler bulunuyor.

model

request

MessageRequest

MessageRequest sınıfı içerisinde bir kullanıcı mesaj gönderdiğinde alınması gereken parametreler bulunuyor.

Login ve Message Request'lerinin ve Response'larının iki farklı request olmasının sebebi ikisi içerisinde farklı işlemler yapabilmemize olanak sağlaması. Dto'daki benzer sebepler burada da geçerli. Bunlar dışında Request ve Response'larımızı ayırmamızın bir sebebi de örneğin kullanıcı giriş yaparken şifre bilgisiyle de giriş yaptı, burada kullanıcının şifre bilgisini de geri dönmek yanlış olur. Bu yüzden Request ve Response sınıfları sadece gerekli bilgileri alıp bu bilgileri geri dönmemize yardımcı oluyor.

model

↳ **request**

↳ **converter**

↳ **LoginRequestConverter**

LoginRequestConverter sınıfı, gelen **LoginRequest**'i **MessageDto**'ya dönüştüren bir sınıf.

model

↳ **request**

↳ **converter**

↳ **MessageRequestConverter**

MessageRequestConverter sınıfı, gelen **MessageRequest**'i **MessageDto**'ya dönüştüren bir sınıf.

Converter'lara ihtiyaç duymamızın sebebi, kullanıcı bir request gönderdi ve biz de arka planda bir takım işlemler yapmak istiyoruz. Bu converter classında kullanıcıdan almadığımız veya almak istemediğimiz bir takım parametreleri alabilir veya işlemler gerçekleştirebiliriz.

model

response

LoginResponse

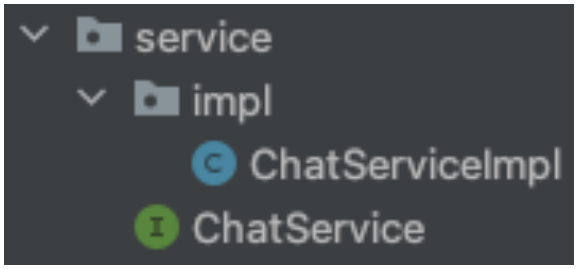
LoginResponse sınıfı içerisinde bir kullanıcı giriş yaptıktan sonra geri dönülmesi gereken parametreler bulunuyor.

model

response

MessageResponse

MessageResponse sınıfı içerisinde bir kullanıcı mesaj gönderdikten sonra geri dönülmesi gereken parametreler bulunuyor.



Service

impl

service

↘

ChatServiceImpl

ChatService interface'i, **Controller** sınıflarında kullanacağımız methodları belirlediğimiz interface diyebiliriz.

service

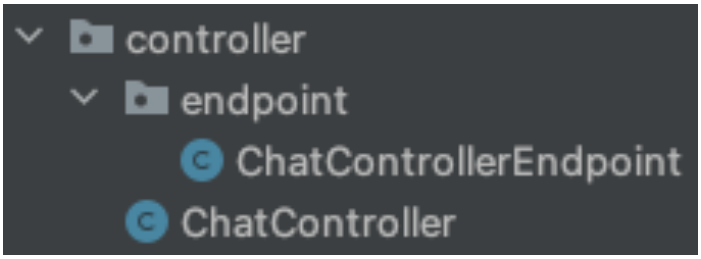
↘

impl

↘

ChatServiceImpl

ChatServiceImpl sınıfı **ChatService** interface'ini implement ediyor ve **ChatService** içerisinde tanımladığımız methodları **ChatServiceImpl** sınıfı içerisinde dolduruyoruz.



Controller

Endpoint

controller

↘

ChatController

ChatController sınıfı, aslında adını da verdiğimiz gibi kontrollerin gerçekleştirildiği sınıf. Yani bu sınıfta, bir method belirliyoruz bu method'un path'i oluyor, parametleri oluyor ve yazdığımız method'a istek geliyor, gelen isteğe göre işlemler yapılarak bir sonuç dönüyor. Biz ise burada tanımladığımız methoda gelen isteği **ChatService** interface'indeki gerekli method'a yönlendiriyoruz ve buradan geri dönen değeri return ediyoruz.

controller

↘

endpoint

↘

ChatControllerEndpoint

ChatController sınıfında kullanılan Endpointlerin kontrol edilebilirliğini sağlamak ve HardCore dediğimiz sabit kodlamadan kaçınmak için oluşturulan ve static final değerlerin tutulduğu bir sınıftır.

- Front-End ile ilgili araştırmalar yapılıp, mimari belirlendi.
- HTML, CSS ve JavaScript kullanımı ile ilgili araştırmalar yapıldı, bir takım denemeler gerçekleştirildi.
- Projeyi canlı bir şekilde herkesin kullanabileceği bir platform oluşturmak için araştırmalar yapıldı ve denemeler gerçekleştirildi.

Bu Süreçten Sonra Yapılacaklar;

- Front-End geliştirmelerinin tamamlanması.
- Projenin demo denemelerinin gerçekleştirilmesi, başka kullanıcılar tarafından denenip eksiklerin görülmesi ve bu eksiklerin tamamlanması.
- Canlı olarak herkesin tüm cihazlarının tarayıcılarından erişebileceği bir platform hâline getirilmesi.

https://github.com/agitrubard/web-chat-app