

1. **REGISTER & LOGIN**

UserController.java



Burada template ile kullanıcının bilgilerini alıp, UserServiceInterface arayüzü ile kullanıcıyı sisteme kaydediyorum.

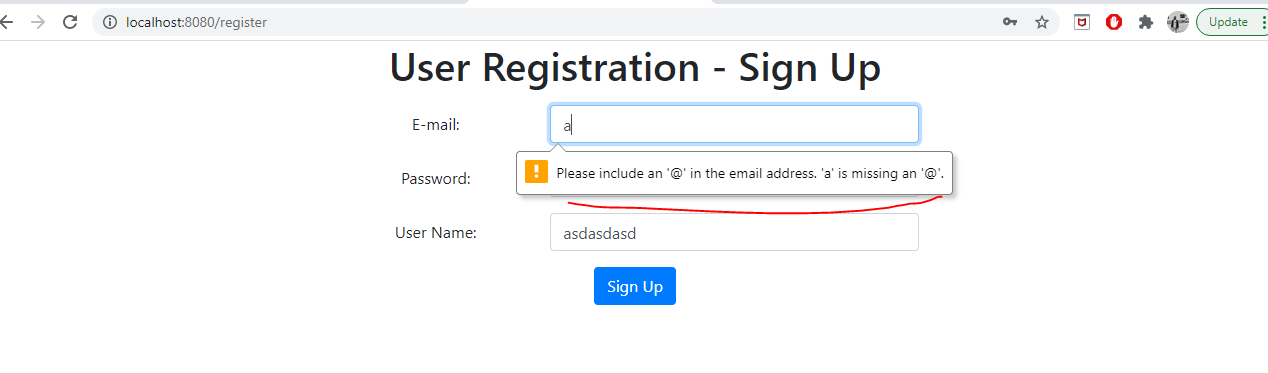
Daha sonra kullanıcı’dan login olması bekleniyor. Start to Chat kısmına login olmadan gidilemiyor. Bunu da bir configürasyon ile sağlıyoruz:

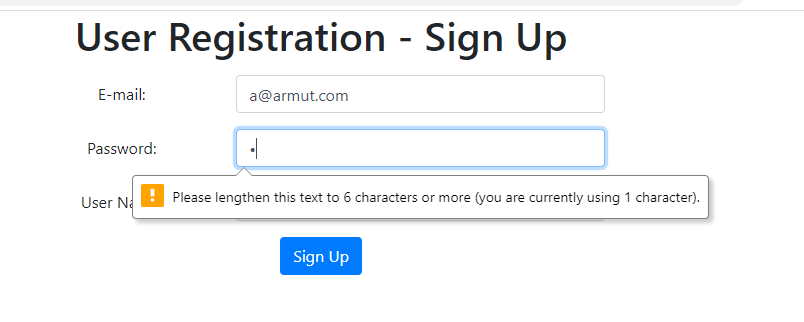
WebSecurityConfig.java



**Index :**   

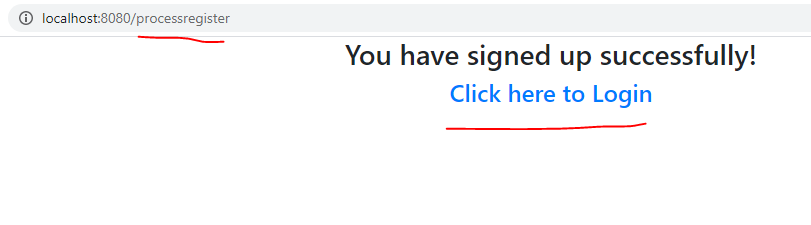

**Registration:**



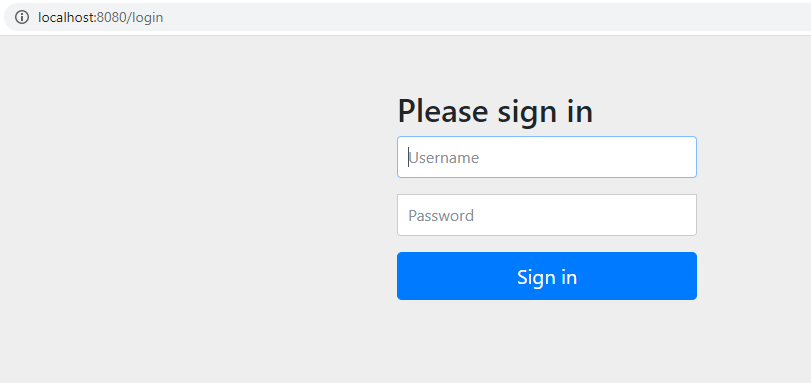


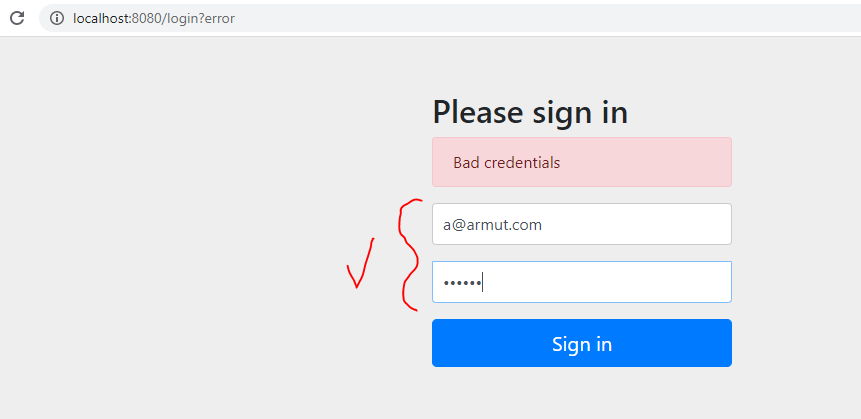


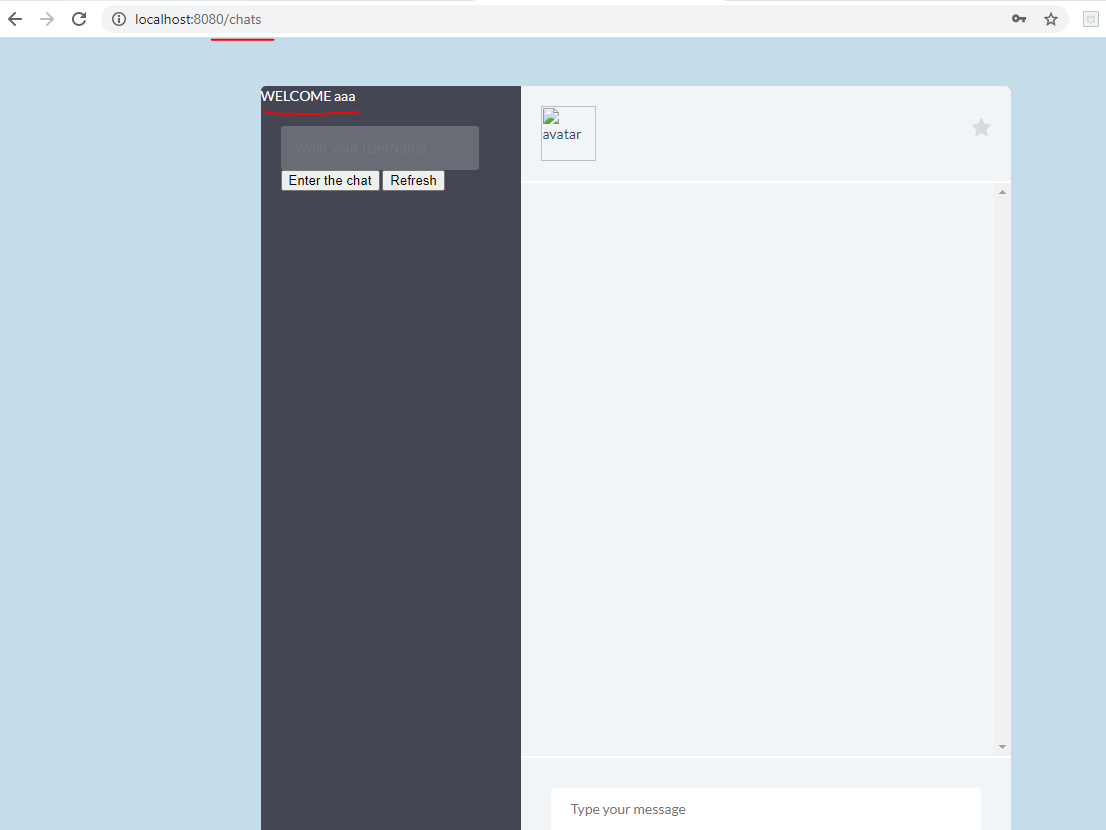
Doğru şekilde register olduktan sonra:



LOGIN Kısmı



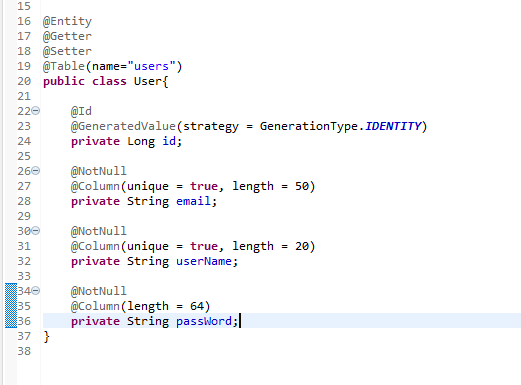




Login olduktan sonra Chat sayfasına geliyoruz.  
  
**SERVICE KISMI**

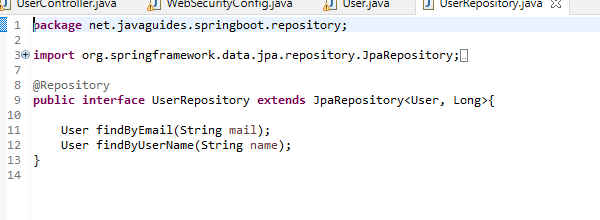
1. User entity oluştur

User.java



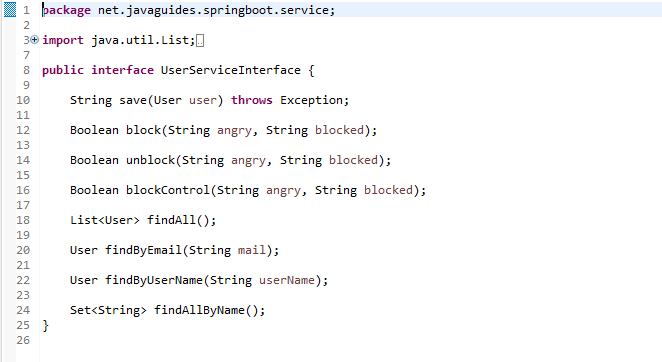
1. User repository oluştur

UserRepository.java



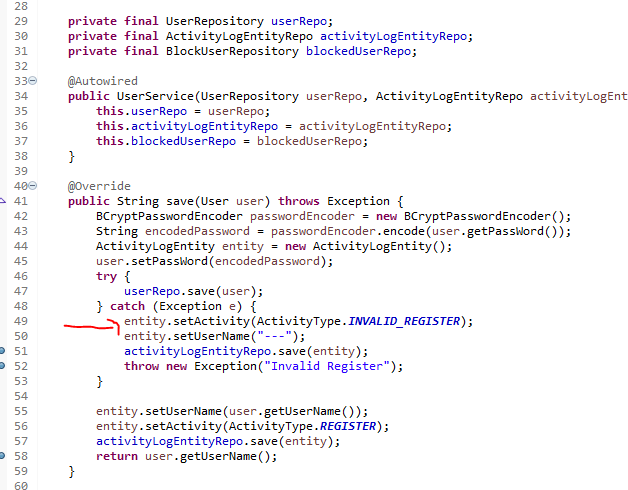
1. UserServiceInterface yaratılır ve metodlar tanımlanır

UserServiceInterface.java



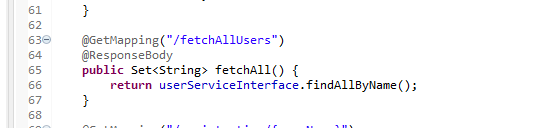
1. UserService yazılır ve interface’de tanımladığımız metodları doldururuz.

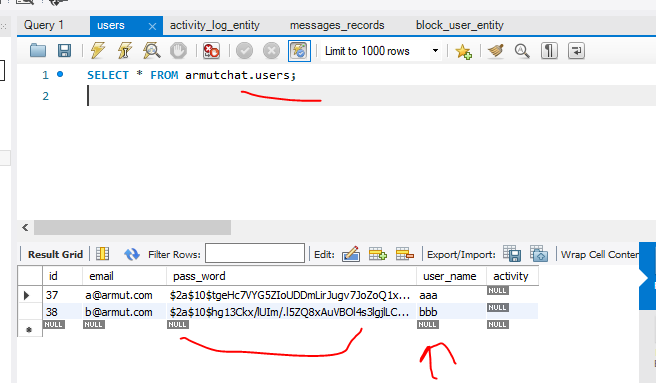
UserService.java

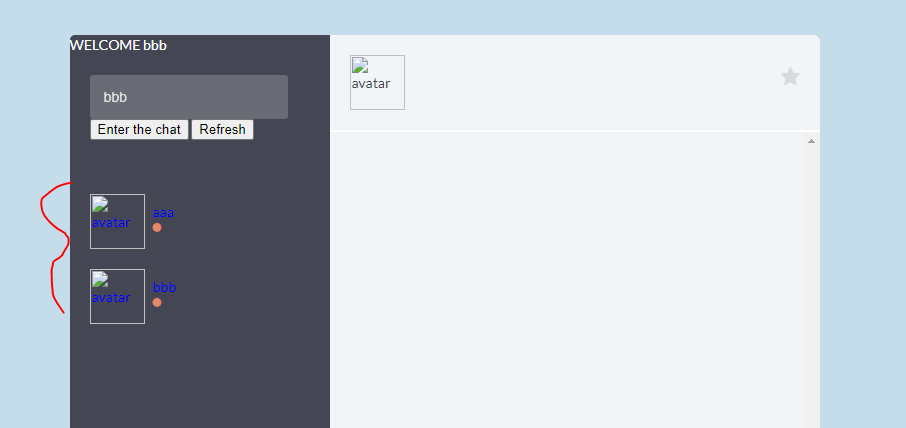


1. **CHAT KISMI**

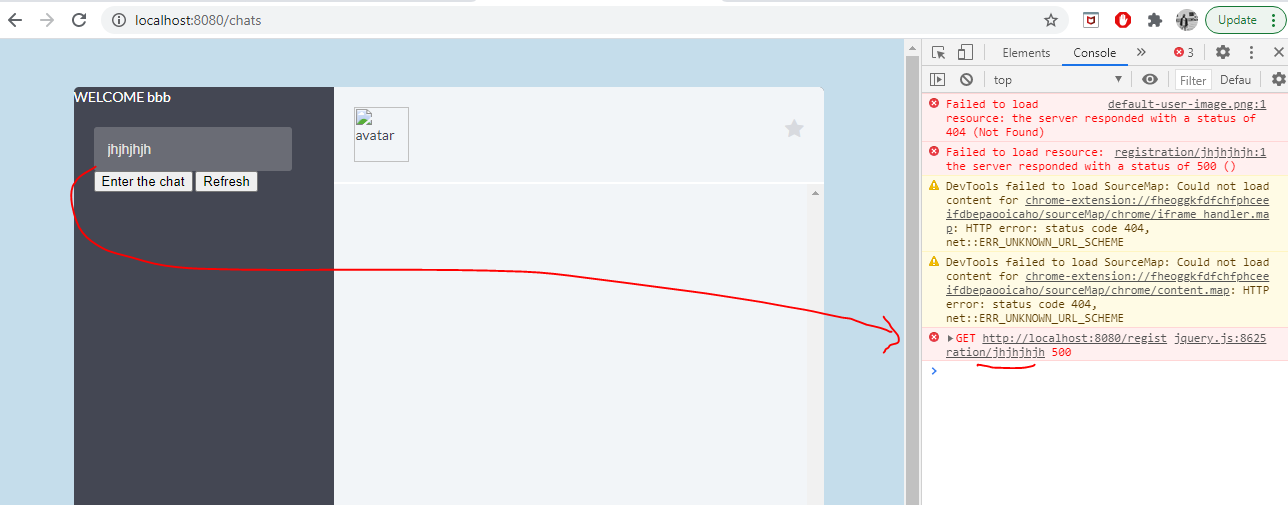
Kullanıcılar birbirlerinin adını bildiği sürece chatleşebilir.  
Ben burada herhangi bir filtreleme yapmadım. Kullanıcılar birbirini engellemediği takdirde karşı tarafa mesaj atabiliyorlar.  
Chat kısmında bütün kullanıcıların listesini alıyorum:  
UserController.java



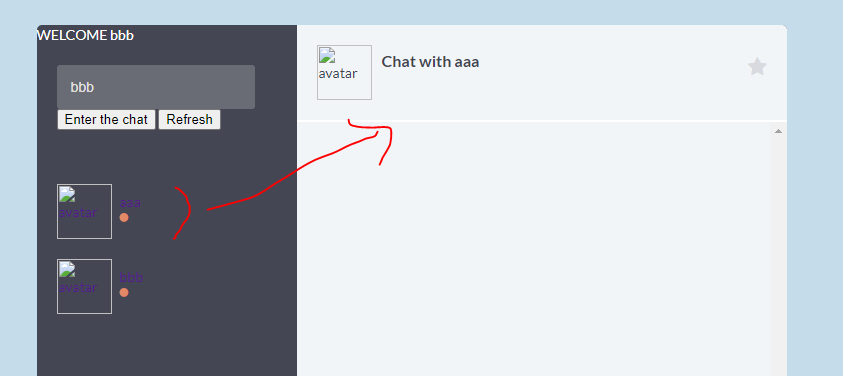
Şifreler encrypted şekilde.  
Register olmuş kullanıcıların kayıtları.



Bu kısımda Eğer db’de kayıtlı olmayan bir ismi girecek olursak chat kısmına giriş yapılamıyor:



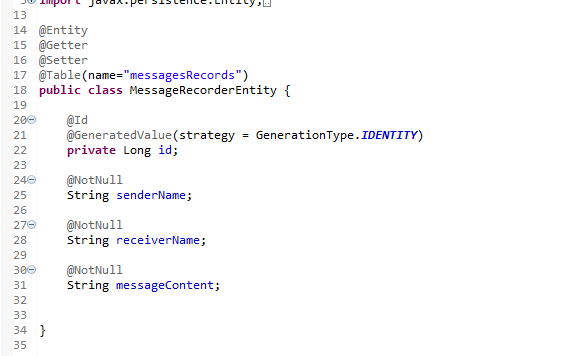
Doğru kullanıcı ile girip Refresh yaptığımızda register olmuş kullanıcılar geliyor:



Chat kısmında karşı tarafa mesaj gönderebiliyoruz.

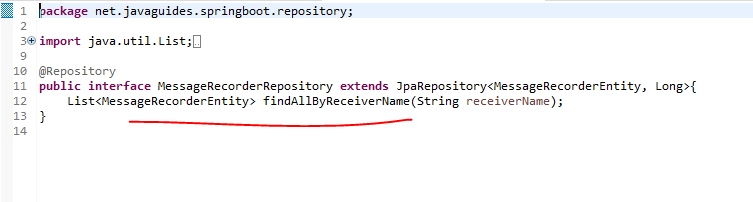
Bize atılan mesajlara DB’den bakacağız.

1. **Geçmişe yönelik mesajlara erişme**
2. Mesajların tutulduğu DB için entity yarattım.

MessageRecorderEntity.java  


1. Mesajlar için Repository interface yarattım:

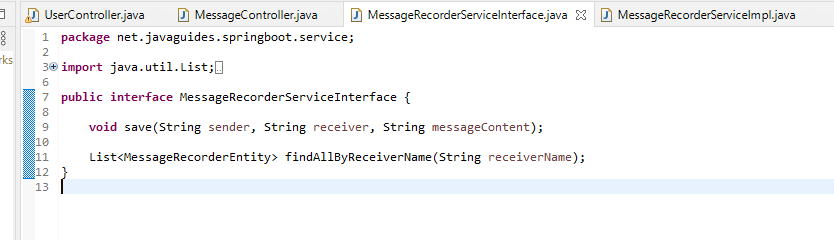
MessageRecorderRepository.java



Içerisinde mesajı alan kişinin adını parameter olarak alan ve o parametreye yollanmış bütün mesajları çeken bir metod tanımlı.  
JPA özelliği sayesinde query yazmamıza gerek kalmadan doğru metod ismiyle verilerimizi database’den çekebiliyoruz.

1. Mesaj kaydı için ServiceInterface ve ServiceImpl:

MessageRecorderServiceInterface.java

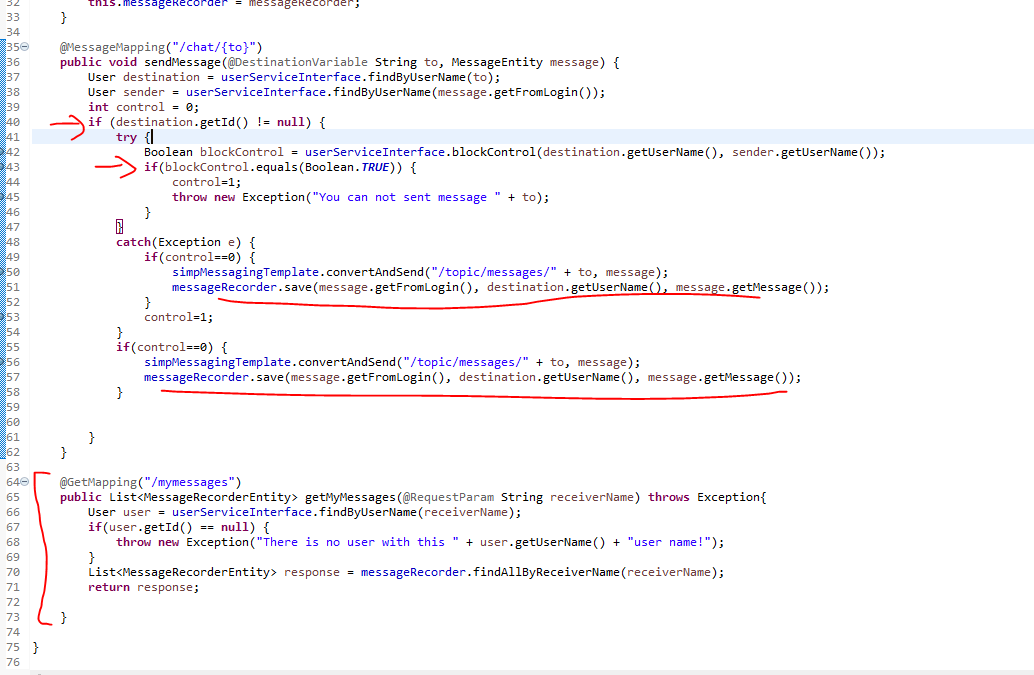


MessageRecorderServiceImpl.java



Burada mesajları hem kayıt ediyoruz hem de gerektiğinde bütün mesajları çekecek metodumuzu yazıyoruz.

1. Mesaj API

MessageController.java  


Mesaj gönderme işlemini yapan api için bir configürasyon yazmıştık:

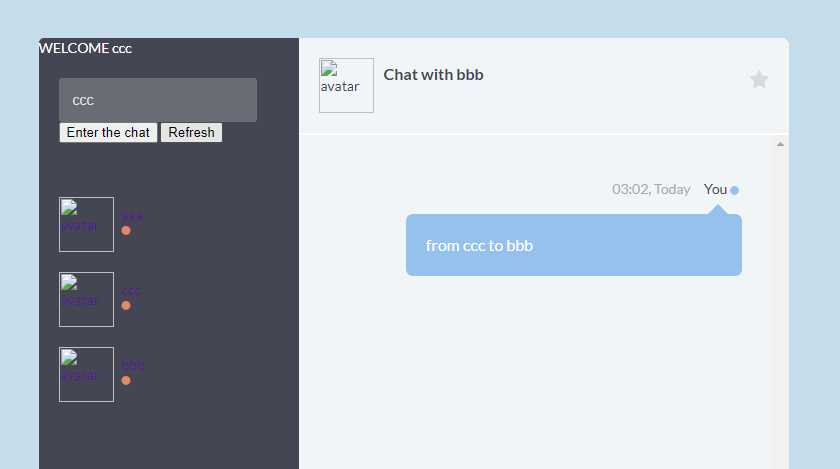
WebSocketConfiguration.java

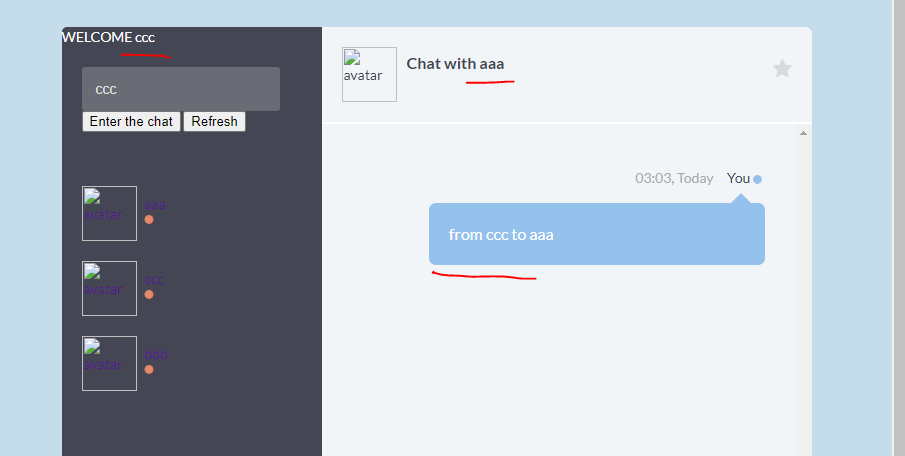


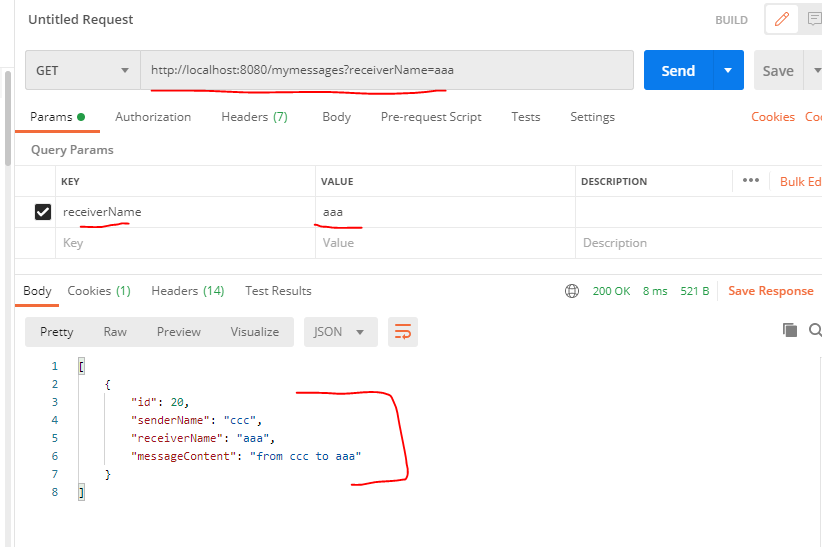
Bu configürasyon websocket kullanımına örnektir.

Biz api içine kendi istediğimiz kontrolleri de kattık görüldüğü üzere. Gönderilen kullanıcının var olup olmadığını ve engel(block) olup olmadığını control ediyoruz. sendMessage()

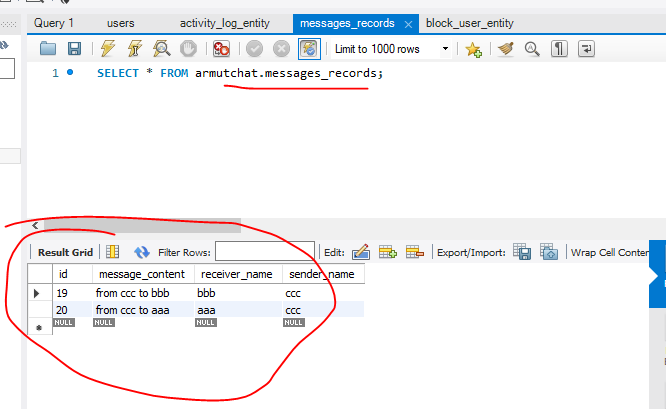
List<MessageRecorderEntity> getMyMessages() Ile de mesajları dönüyoruz.  
  
  
  
  
bir örnek gösterelim:





ccc kullanıcısından aaa ve bbb kullanıcılarına mesaj attık.  


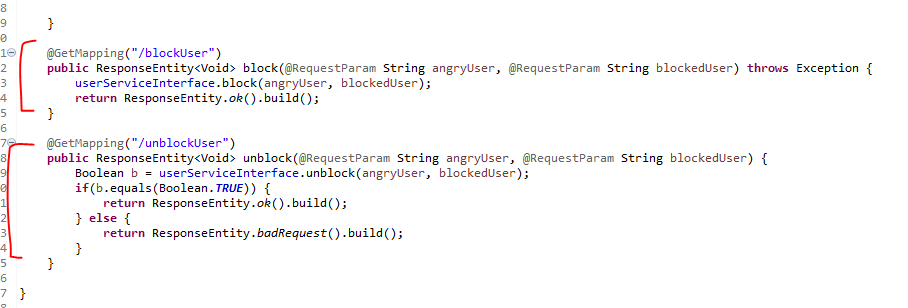
<http://localhost:8080/mymessages?receiverName=aaa>



Görüldüğü üzere mesajlarımızı kaydediyoruz. Istediğimiz zaman da <http://localhost:8080/mymessages?receiverName=aaa>

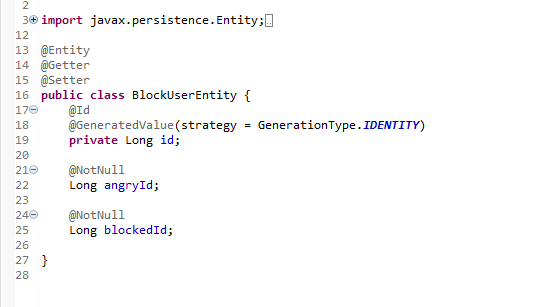
Ile mesajlarımıza erişebiliriz.

1. **KULLANICILARIN BİRBİRİNİ ENGELLEMESİ VE ENGELİ KALDIRMA İŞLEMİ**

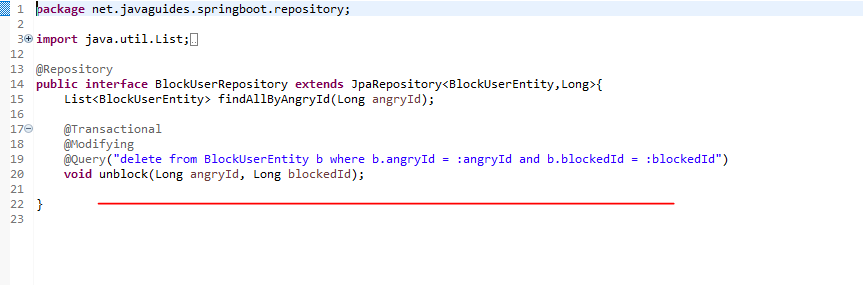
UserController.java  


bu işlemin yapılabilmesi için user id’lerin tutulduğu bir tablo ihtiyacım vardı. Yine genel spring boot mimarisine uygun olarak bir entity yarattım:

BlockUserEntity.java



BlockUserRepository.java



Bu sefer kendime query annotation yazmam gerekti çünkü bir kullanıcı birden fazla kullanıcıyı engelleyebilir.

Benim silmem gereken satır için iki parameter kullanmam gerekli(engel kaldırmak için.).

Sadece userId’leri tutmam yeterli çünkü user’ları findByUserId ile bulabiliyorum.

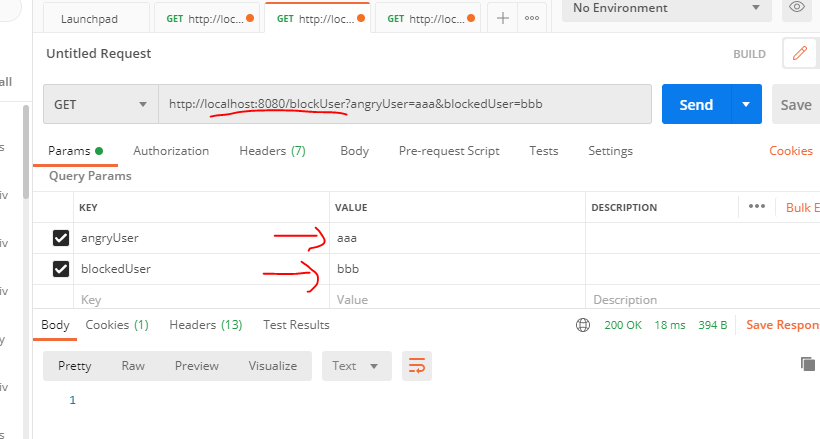
Silme işlemleri için @Transactional @Modifying anotationlarını kullanmam gerekiyor. Aksi takdirde izin vermiyor Hibernate.

UserService.java

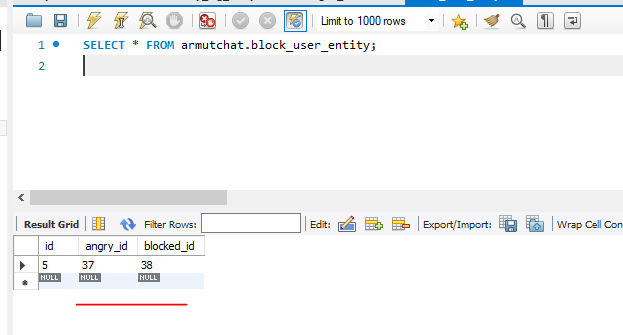


Bu işlemler için ayrıca bir servis ve interface yaratmıyorum.

Örnek yapalım:

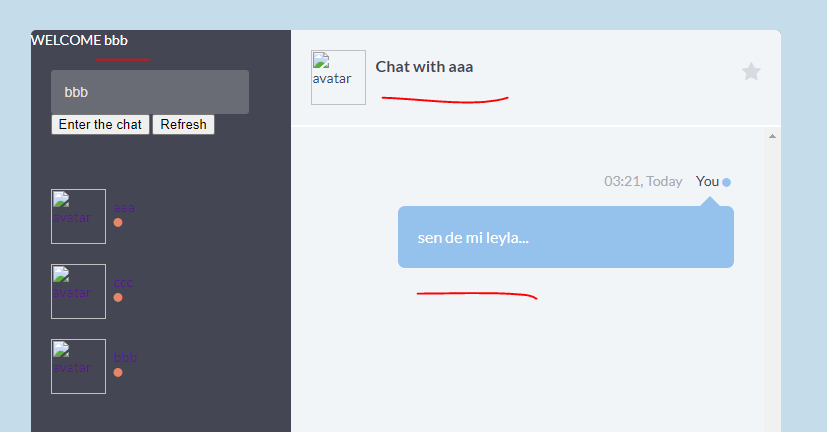


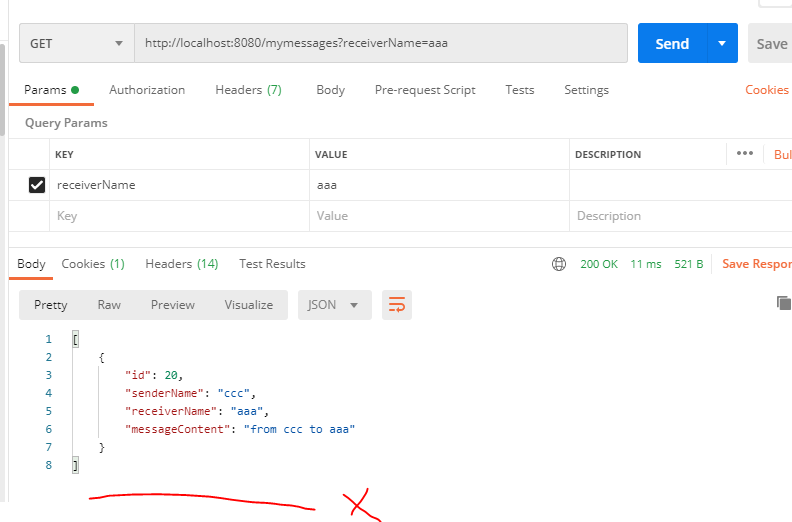
<http://localhost:8080/blockUser?angryUser=aaa&blockedUser=bbb>



aaa kullanıcı olarak bbb kullanıcısını engelledik.

Şimdi bbb’den aaa’ya mesaj atalım:



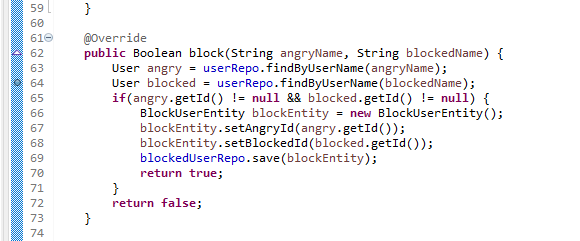


<http://localhost:8080/mymessages?receiverName=aaa>

böylelikle geçmişteki mesajlarına eriştiğini tekrardan test etmiş olduk.

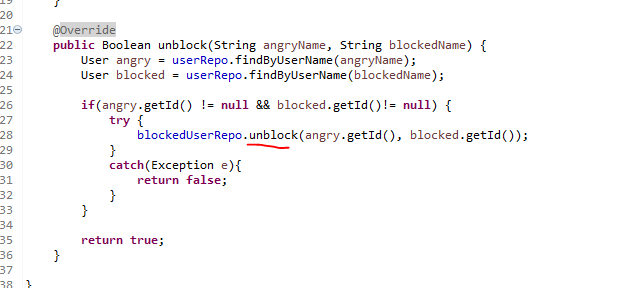
Servislerimi gösteriyorum:

UserService.java

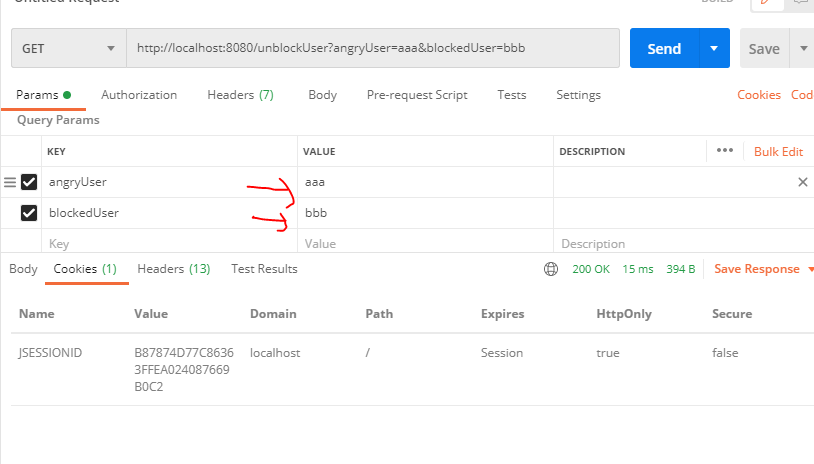




Mesajı göndermeden önce kullanıcının bloklu olup olmadığını da bu metod ile control ediyorum.

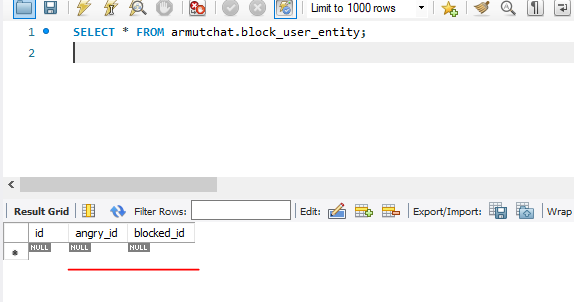


Bu da query yazdığımız bloklu kaydı silmek için yazıdğım metod.

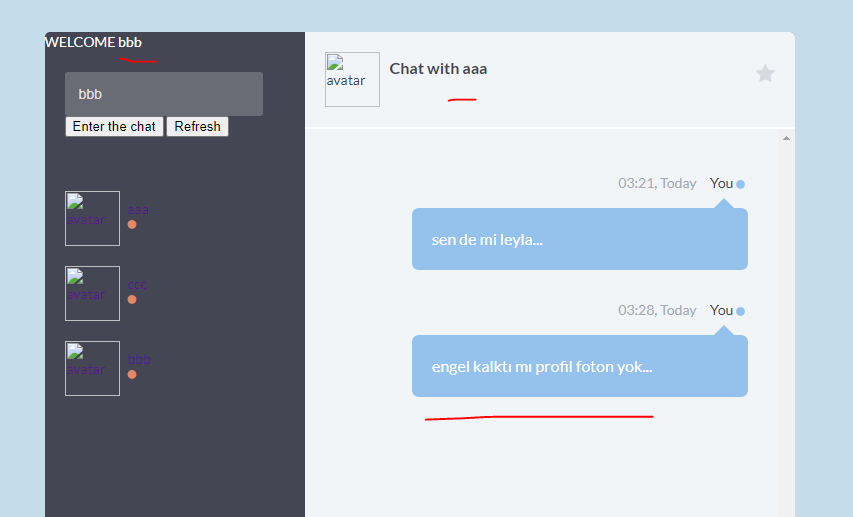


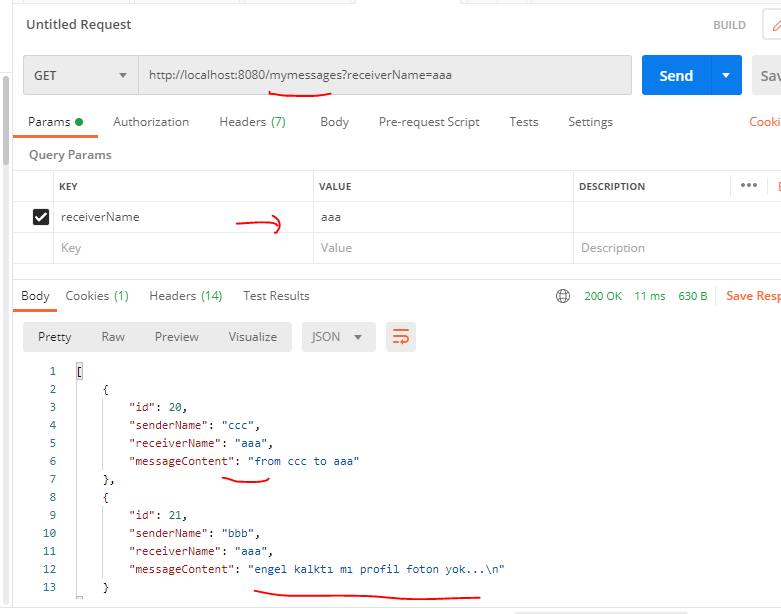
<http://localhost:8080/unblockUser?angryUser=aaa&blockedUser=bbb>

bloğu kaldıralım ve DB’yi control edelim:



Şimdi tekrardan mesaj atmaya çalışalım ve mesaj geçmişinden control edelim:



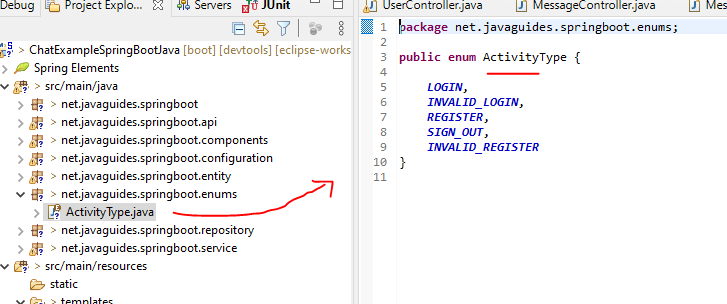


Engeli kaldırdıktan sonra mesajlarımız karşı tarafa gitmeye devam ediyor…

1. **AKTIVITY LOGLARININ TUTULMASI**

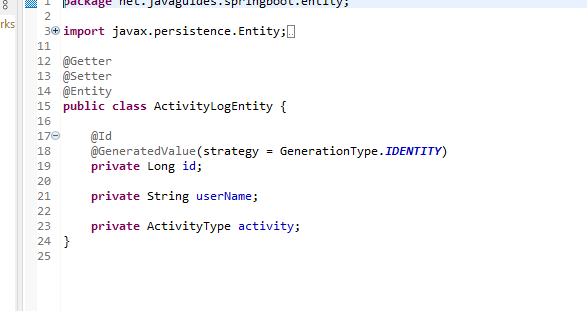
Bu logların tutulması için bir enum sınıfı yarattım:

ActivityType.java



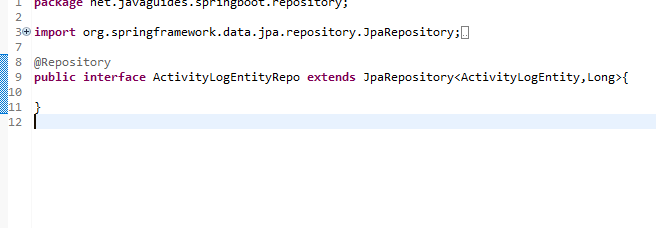
Tabi ki entity’e de ihtiyacım var kayıtların tutulabilmesi için:

ActivityLogEntity.java



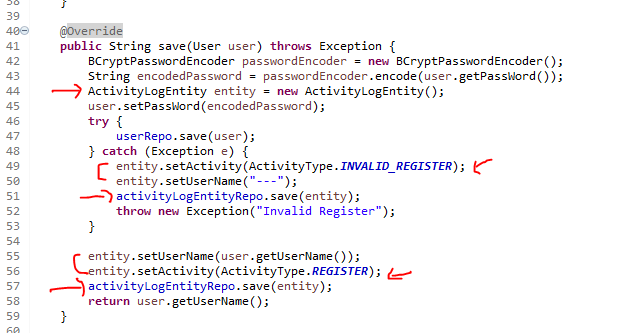
Ve Repository’im:

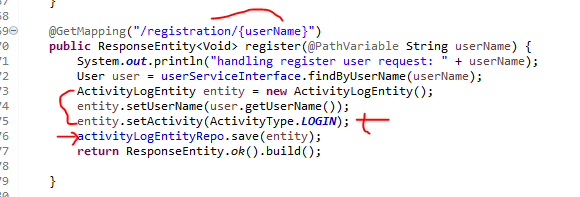
ActivityLogEntityRepo.java



Örnek bir kaç kullanım göstereceğim:

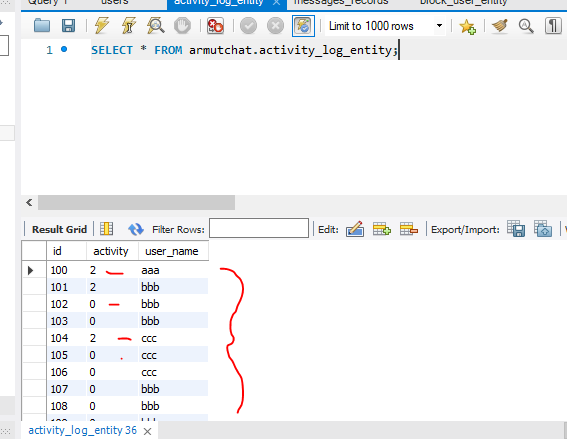
UserService.java



UserController.java  


Bu api chat’e kullanıcı adı girip online olmamızı sağlayan api.

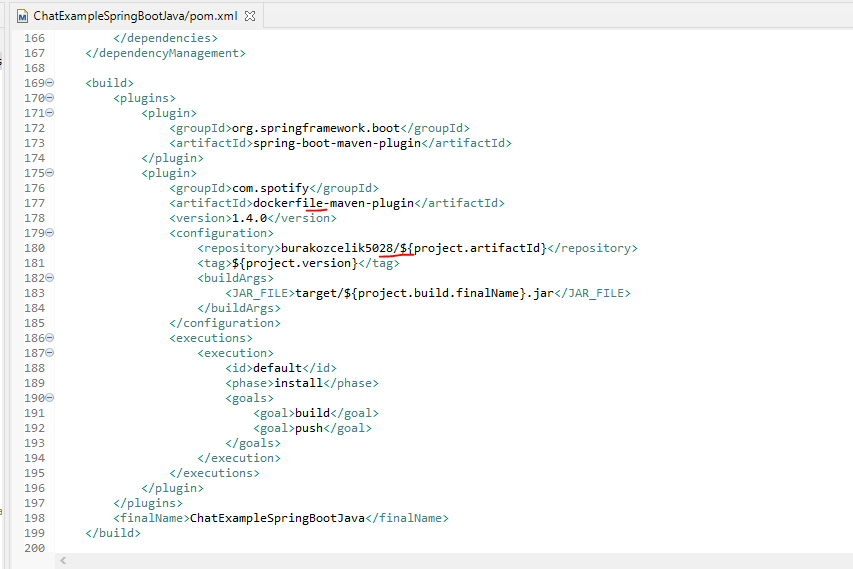
Bunu da böylelikle paylaşmış oldum. Bizim ana register’dan farklı. Enter Chat kısmı için sadece.



Login ve Registration loglarımızı tutuyoruz.

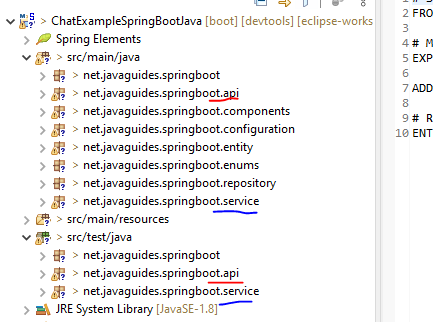
**DOCKERIZE ETME ADIMLARI**

Pom.xml





**Unit Testler**



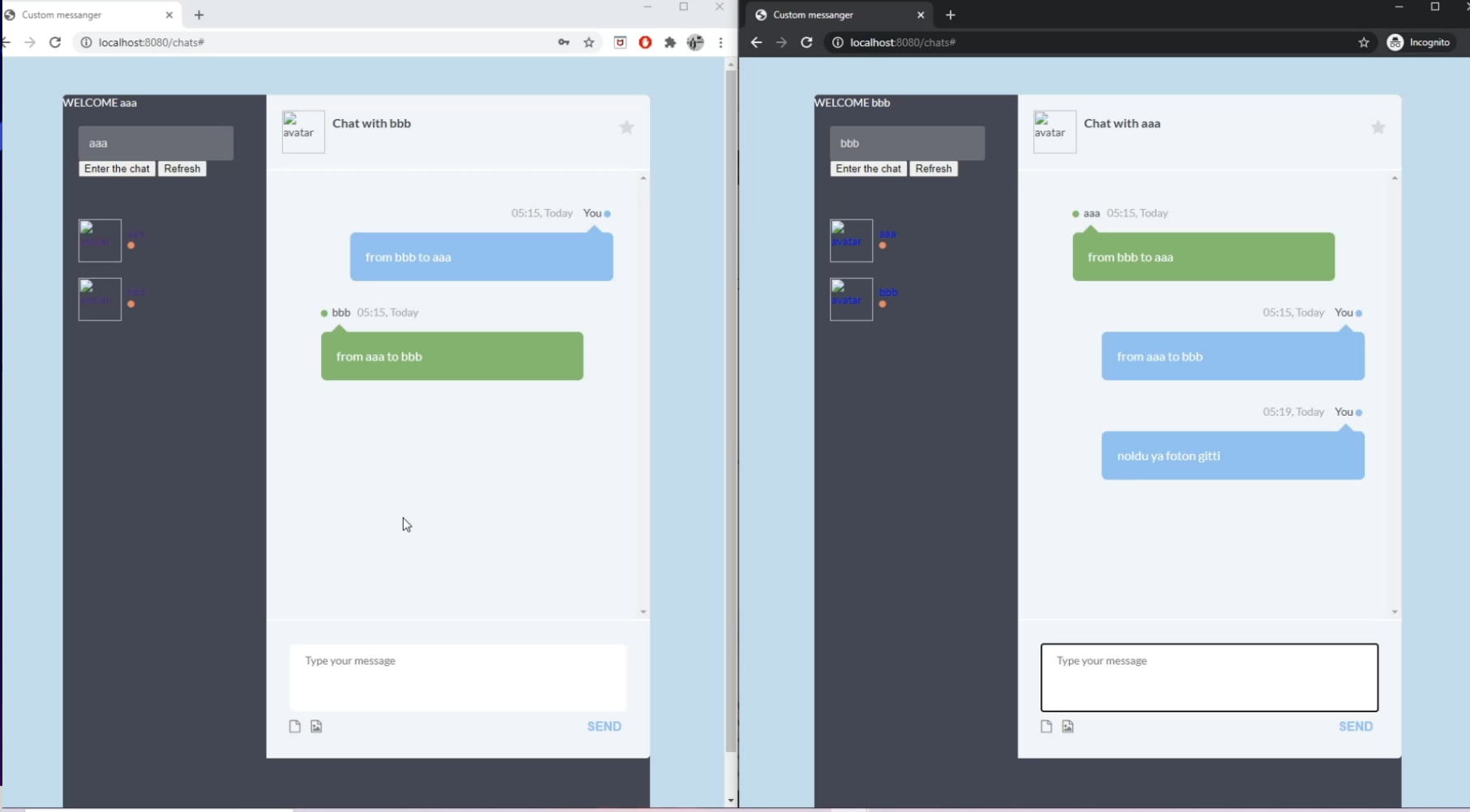
Testler ile testleri yazılan sınıfları ikiz uzantılar altında tutuyoruz.

MessageRecorderServiceImplUnitTest.java



Servislerimizi ve repositorylerimizi mocklamamız gerekiyor.

Metodların başına, classisimlerinin sonuna Test yazmamız idealdir.



Mesajları yanlış atmışım burda…