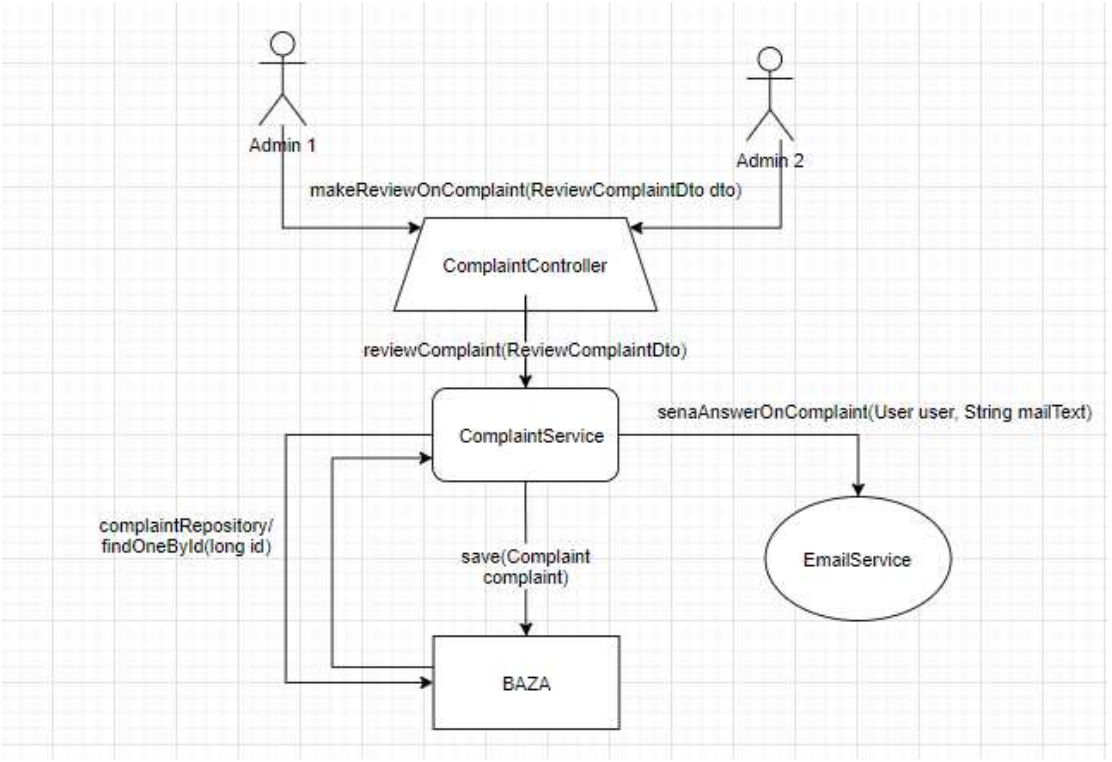
Konfliktne situacije

1. Na jednu zalbu moze da odgovori samo jedan administrator sistema

Opis:

Do konfliktne situacije dolazi kada dva administratora u isto ili preklapajuce vreme pokusaju da odgovore na jednu istu zalbu. Moguci ishod je da u slucaju aplikacije zalba bude dvaputa oznacena kao obradjena I da se podnosiocu zalbe I osobi za koga se zalba odnosi dostave dva razlicita odgovora.

Graf toka informacija:



Resavanje konfliktne situacije:

Za resavanje konfliktne situacije koristi se persimisticko zakljucavanje.
Prva tacka koju klijent odnosno administrator pogadja na serveru je ComplaintController odnosno konkretno sledeca metoda:

```
@PutMapping(path = "/reviewComplaint")
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<?> makeReviewOnComplaint(@RequestBody ReviewComplaintDto dto){
    complaintService.reviewComplaint(dto);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Kao parametar metoda dobija ReviewComplaintDto koji u sebi sadrzi tekst odgovora koji se prosledjuju podnosiocu zalbei osobi za koga je zalba namenjena, pored toga sadrzi I mejlove te dve osobe. Zatim podaci se prosledjuju klasi ComplaintService gde se resava citav problem.

```
@Transactional
public void reviewComplaint(ReviewComplaintDto dto){
    Complaint complaint = complaintRepository.findOneById(dto.getComplaintId());
    if(complaint.isReviewed())
        return;
    complaint.setReviewed(true);
    save(complaint);
    sendMails(userRepository.findByEmail(dto.getCustomerMail()),
            userRepository.findByEmail(dto.getOwnerMail()), dto);
}
```

Metoda reviewComplaint je bazna metoda za resavanje problema I anotirana je sa @Trnsactional, gde je podrazumevna vrednost readOnly paramatra false, jer se u njoj podaci menjaju, a nije potrebno ni otvaranje nove transakcije vec se sve moze odraditi u postojecoj. Kljucna linija koda je poziv metode complaintRepository.findOneById(long id).

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("select c from Complaint c where c.id = :id")
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value ="0")})
Complaint findOneById(@Param("id") long id);
```

U okviru ove metode se konkretn podatak sa prosledjenim id-em zakljucava u bazi I nije moguce niti njegovo citanje niti njegov pisanje dok nit koja ga je prva zauzela ne odrdi svoju transakciju I ne izmeni podatak.

Zatim se vrsi provera da li je konkretan podatak vec revidiran I ako jeste tu se dalja prica zavrsava.
U slucaju da nije konkretna zalba se oznacav kao revidirana sacuvava I vrsi se slanje mejlova, u okviru sledecih metoda:

```
@Transactional
public Complaint save(Complaint complaint) { return complaintRepository.save(complaint); }
```

```
private void sendMails(User customer, User owner, ReviewComplaintDto dto) {
    try {
        emailService.sendAnswerOnComplaint(customer, dto.getMailForCustomer());
        emailService.sendAnswerOnComplaint(owner, dto.getMailForOwner());
    }catch (Exception ignored){ }
}
```

Save metoda je takodje anotiranaj kao transakcija zato sto menja podatke u bazi, dok metoda sendMils poziva emailService I sledecu asinhronu metodu:

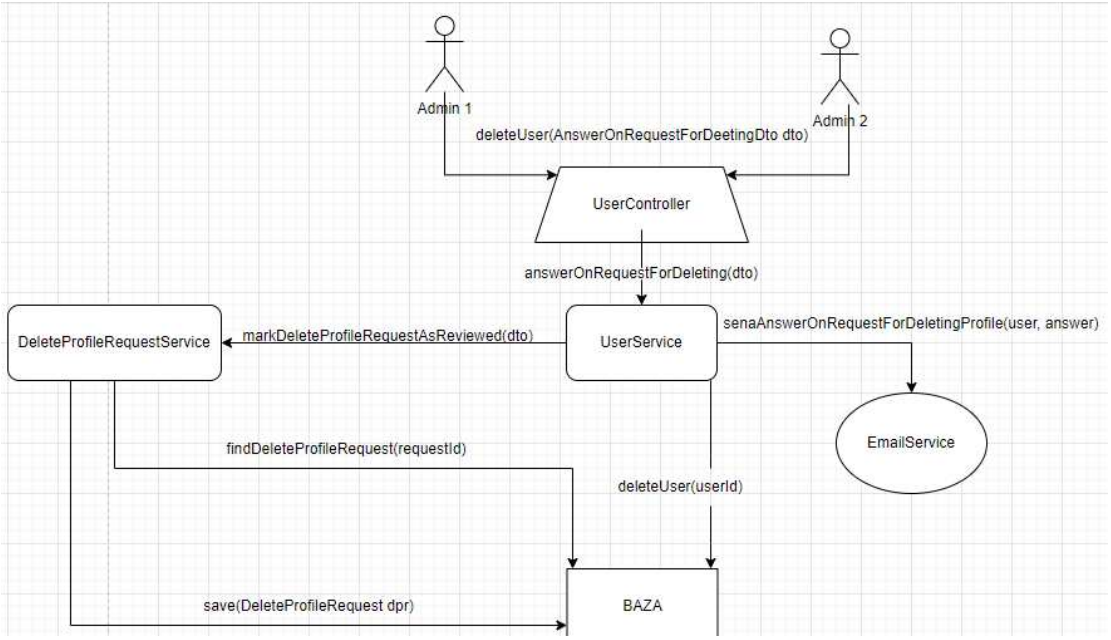
```java
@Async
public void sendAnswerOnComplaint(User user, String answer) throws MailException,
        InterruptedException{
    SimpleMailMessage mail = new SimpleMailMessage();
    mail.setTo(user.getEmail());
    mail.setFrom(env.getProperty("spring.mail.username"));
    mail.setSubject("Obaveštenje o žalbi.");
    mail.setText("Pozdrav " + user.getFirstname() + ", \n\n" +
            "ovom prilikom želimo da vas obavestimo sledeće\n" +
            answer +
            "\n\nS poštovanjem admin tim.");
    javaMailSender.send(mail);
}
```

2. Na jedan zahtev za brisanje naloga moze da odgovori samo jedan administrator sistema

Opis:

Do konfliktne situacije dolazi kada dva administratora u isto ili preklapajuce vreme pokusaju da odgovore na jedan isti zahtev za brisanje. Zbog logike koja je implementirana ukoliko dodje do odobravanja brisanja I sam zahtev ce biti obrisan pa bi u slucaju konflikta doslo do kolizije u sistemu.

Graf toka podataka:



Resavanje konfliktne situacije:

Za resavanje konfliktne situacije koristi se persimisticko zakljucavanje.
Nakon sto korisnik pogodi UserController sa svojim odgovorom na zahtev za brisanje podaci se dalje prosledjuju na obradu unutar UserService klase. Metoda koja se poziva je answerOnRequetsForDeleting koja je anotirana sa @Transactional.

```
@Transactional
public void answerOnRequestForDeleting(AnswerOnRequestForDeletingDto dto) {
    DeleteProfileRequest request = deleteProfileRequestService.markDeleteProfileRequestAsReviewed(dto.getRequestId());

    if(request == null)
        return;

    if(dto.isForDelete())
        deleteUser(request.getUser().getId());

    try { emailService.sendAnswerOnRequestForDeletingProfile(request.getUser(), dto.getAnswer());
    }catch (Exception e){ System.out.println(e);}
```

Zahtev se dalje salje na obradu u okviru DeleteProfilRequestService klase, metoda markDeleterofileRequestAsReviewed koja otvara potpuno novu transakciju:

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public DeleteProfileRequest markDeleteProfileRequestAsReviewed(long id){
    DeleteProfileRequest request = deleteProfileRequestRepository.findOneById(id);
    if(request.isReviewed())
        return null;
    request.setReviewed(true);
    return save(request);
}
```

Bitna linija koda je poziv metode deleteProfileRequestRepository.findOneById:

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("select dpr from DeleteProfileRequest dpr where dpr.id = :id")
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value ="0")})
DeleteProfileRequest findOneById(@Param("id") long id);
```

U okviru ove metode se konkretn podatak sa prosledjenim id-em zakljucava u bazi I nije moguce niti njegovo citanje niti njegov pisanje dok nit koja ga je prva zauzela ne odrdi svoju transakciju I ne izmeni podatak.

Metoda save radi izmenu konkretnog podatka u bazi:

```
@Transactional
public DeleteProfileRequest save(DeleteProfileRequest deleteProfileRequest){
    return this.deleteProfileRequestRepository.save(deleteProfileRequest);
}
```

Podatak se ponovo vraca u UserController gde se provera da li uopste postoji u slucaju da je prethodno obrisan. Ako postoji nastavlja se brisanje profila I obavestavanje korisnika o brisanju profila:

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void deleteUser(long id) { userRepository.deleteById(id); }
```

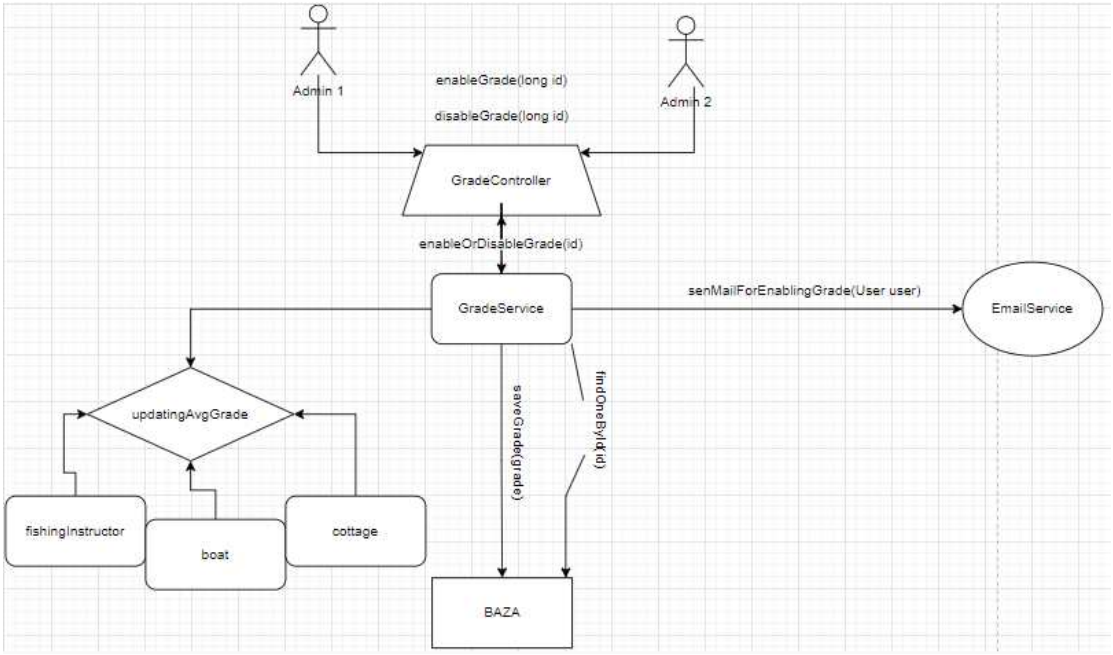
Brisanje profila korisnika koji je podneo zahtev vrsi se u okviru nove transakcije.

3. Odobravanje ocena od strane admina

Opis:

Sistem treba onemoguciti da dva razlicita admina u preklapajucem ili istom vremenu odrade reviziju ocene vezane za neku rezervaciju. Moguci ishod je da jedan admin odobri ocenu, a zatim je drugi admin ponisti I korisnik biva obavesten o odobravanju ocene koja se zapravo nece prikazati I uci u prosecnu ocenu njegove usluge.

Graf toka informacija:



Resavanje konfliktne situacije:

Prva taca koja se gadja od strane admina obuhvata dve metode u klasi GradeControler u zavisnosti da li se ocena odobrava ili odbija, odnosno konkreto metode enableGrade I disableGrade, u okviru obe metode poziva se GradeService sa prametrima id konkretne ocene za procesuiranje I parametra o tretiranju konkretne ocene odnosno da li ce biti odobrena ili odbijena:

```java
@GetMapping(path = "/disable/{id}")
public ResponseEntity<?> disableGrade(@PathVariable long id){
    gradeService.enableOrDisableGrade(id, forEnabling: false);
    return new ResponseEntity<>(HttpStatus.OK);
}

@GetMapping(path = "/enable/{id}")
public ResponseEntity<?> enableGrade(@PathVariable long id){
    gradeService.enableOrDisableGrade(id, forEnabling: true);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Zatim se podaci prosledjuju klasi GradeService gde je kompletna logika odradjena u okviru metode enableOrDisableGrade, metoda je anotirana sa @Trnsactional I kao parametar propagation prosledjeno je otvaranje nove transakcije:

```java
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void enableOrDisableGrade(long gradeId, boolean forEnabling) {
    Grade grade = gradeRepository.findOneById(gradeId);

    if(grade.isReviewed()) return;

    if(forEnabling) grade.setEnabled(true);

    grade.setReviewed(true);

    saveGrade(grade);
    User user = null;
    if(grade.getInstructor() != null) {
        grade.getInstructor().setRating(CalculateNewAverageGradeForInstructor(gradeRepository.findAllGradesOfInstructor(grade.getInstructor().getId())));
        user = fishingInstructorRepository.save(grade.getInstructor());
    }
    else if (grade.getCottage() != null) {
        grade.getCottage().setRating(CalculateNewAverageGradeForCottage(gradeRepository.findAllGradesOfCottage(grade.getCottage().getId())));
        user = cottageRepository.save(grade.getCottage()).getCottageOwner();
    }
    else if (grade.getBoat() != null) {
        grade.getBoat().setRating(CalculateNewAverageGradeForBoat(gradeRepository.findAllGradesOfBoat(grade.getBoat().getId())));
        user = boatRepository.save(grade.getBoat()).getBoatOwner();
    }
```

Problem je resen persimistickim zakljucavanjem u okviru metode findOneById koja je implementirana u interfejsu GradeRepository, tu se konkretan red u tabeli vezan za ocenu odnosno podatak sa prosledjenim id-em zakljucava I za citanje I za pisanje dok se transakcija ne zavrsi:

```java
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("select g from Grade g where g.id = :id")
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value ="0")})
Grade findOneById(@Param("id") long id);
```

Dalje nastavlje se klasican tok azuriranja ocene I obavestavanje vlasnika rezervacije na koju se ocena odnosi da je ocena odobrena ili u suprotnom se ne desava nista ukoiko ocena nije odobrena.