



MIDDLE EAST TECHNICAL UNIVERSITY
NORTHERN CYPRUS CAMPUS

CNG 495

FALL - 2023

TERM PROJECT FINAL REPORT

Team Members: Bartu Can PALAMUT, Doğukan AKDAĞ, Nurberat
GÖKMEN

ID Numbers: 2386225, 2452928, 2453231

Project Name: PhotoHocam

Table of Contents

1. Introduction	5
1.1 Project Brief	5
2. Project Description	6
2.1 User Manual	6
2.2 Project Activity Diagrams	6
2.3 Parts of the project	8
2.3.1 Back-end side	8
2.3.2 Front-end side	26
3. Statistics	37
4. References	39

Table of Figures

Figure 1 Activity Diagram	6
Figure 2 Activity Diagram	7
Figure 3 User Entity	8
Figure 4 User Repository	8
Figure 5 UserDetails	9
Figure 6 JWT Provider / Generator	10
Figure 7 Authentication Filtering	10
Figure 8 Authentication Entry Point	11
Figure 9 Security Config	11
Figure 10 FriendRequest Entity	12
Figure 11 FriendRequest Repository	13
Figure 12 sendFriendRequest service	13
Figure 13 approveFriendRequest service	14
Figure 14 Image Entity	15
Figure 15 Image Repository	16
Figure 16 Send Image Service	17
Figure 17 Delete Image Service	18
Figure 18 ImageDto Class	18
Figure 19 Get Image List Service	19
Figure 20 FriendRequest Repository	20
Figure 21 approveFriendRequest service	21
Figure 22 Image Entity	22
Figure 23 Send Image Service	23
Figure 24 Delete Image Service	24
Figure 25 ImageDto Class	24
Figure 26 Get Image List Service	25
Figure 27 Project Structure	26
Figure 28 Constants Class	26
Figure 29 Camera Provider	27
Figure 30 ApiService Class	28
Figure 31 Api Calls	29
Figure 32 Api Calls	30
Figure 33 Api Calls	30

Figure 34 UI of Authentication	31
Figure 35 UI for Camera	32
Figure 36 UI for Un/Friends	33
Figure 37 UI for Messages	34
Figure 38 Api Call	35
Figure 39 Api Call	36
Figure 40 Commit History	37
Figure 41 Individual Commits	37
Figure 42 SQL Cloud CPU Utilization	37
Figure 43 App Engine Memory Usage	38
Figure 44 App Engine Traffic	38

1. Introduction

1.1 Project Brief

Our project's purpose is to create a dynamic photo-sharing application that prioritizes user privacy and convenience. Users can register, send friend requests, and engage in private chats where they can securely share photos. To enhance privacy, received photos can only be viewed once. We've implemented a database to store photos in byte format until accessed, ensuring both accessibility and security. The backend server is developed in Java with Spring Boot, utilizing Software as a Service (SaaS) with Google Cloud, specifically App Engine and Cloud SQL, to streamline infrastructure management. The client side is a Flutter application, ensuring easy updates without user intervention.

This project offers several benefits, including a user-friendly platform for photo sharing with enhanced privacy features. The approach of storing images as bytes in the database and converting them to base64 contributes to improved security and accessibility. Use of SaaS with Google Cloud simplifies infrastructure management, allowing the development team to focus exclusively on enhancing user experiences. The Flutter application ensures easy updates without requiring users to manually update their applications, reducing maintenance challenges.

The novelty of our project is in our approach to image storage. By storing images as bytes in the database and converting them to base64 for frontend display, we address security and accessibility concerns in a novel way. Combined with the utilization of Google Cloud's advanced SaaS model, this contributes to a more streamlined and secure user experience.

Widely used projects similar to ours include Snapchat, Instagram, and other photo-sharing platforms. However, our one-time viewability and the innovative image storage method sets our project apart, providing an additional layer of privacy for users.

GitHub Repository and Project video is available on references 8 and 9 respectively.

2. Project Description

2.1 User Manual

User can register or login from the splash / welcoming page when s/he opens the application. If register / login process will successful, user will be navigated to camera page. User can logout of the system with clicking “Exit” button which is available on top left of the application. User can navigate user list screen for adding new friends by clicking “+” button which is available on top right of the application. User can navigate friend request list screen for approving friend requests by clicking “person” button which is available next to the “+” button. User can navigate to the messages page by clicking “messages” button which is available next to the “person” button. User can take a photo by clicking big button which is available bottom of the application. If user clicks that button, photo will appear on the screen. If user clicks “send” button on that screen, user’s friend list will appear on the screen, and user will choose one of friends to send that photo. If user received any photo, s/he can see them from messages page.

2.2 Project Activity Diagrams

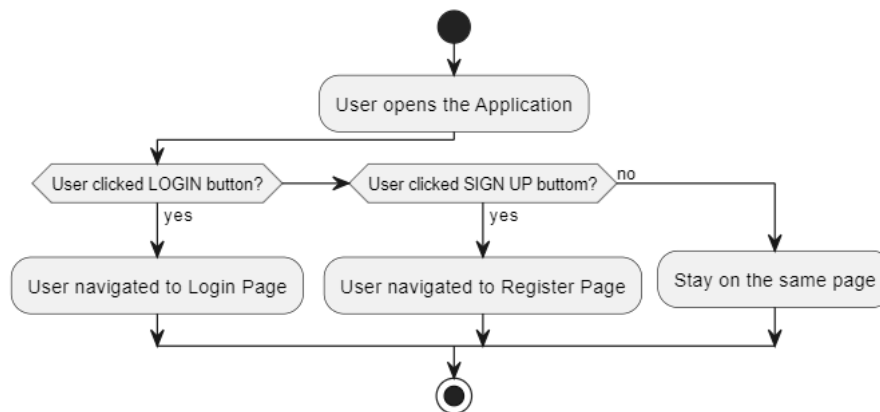


Figure 1 Activity Diagram

Figure 1 shows the splash page’s flow chart of our client side of the project.

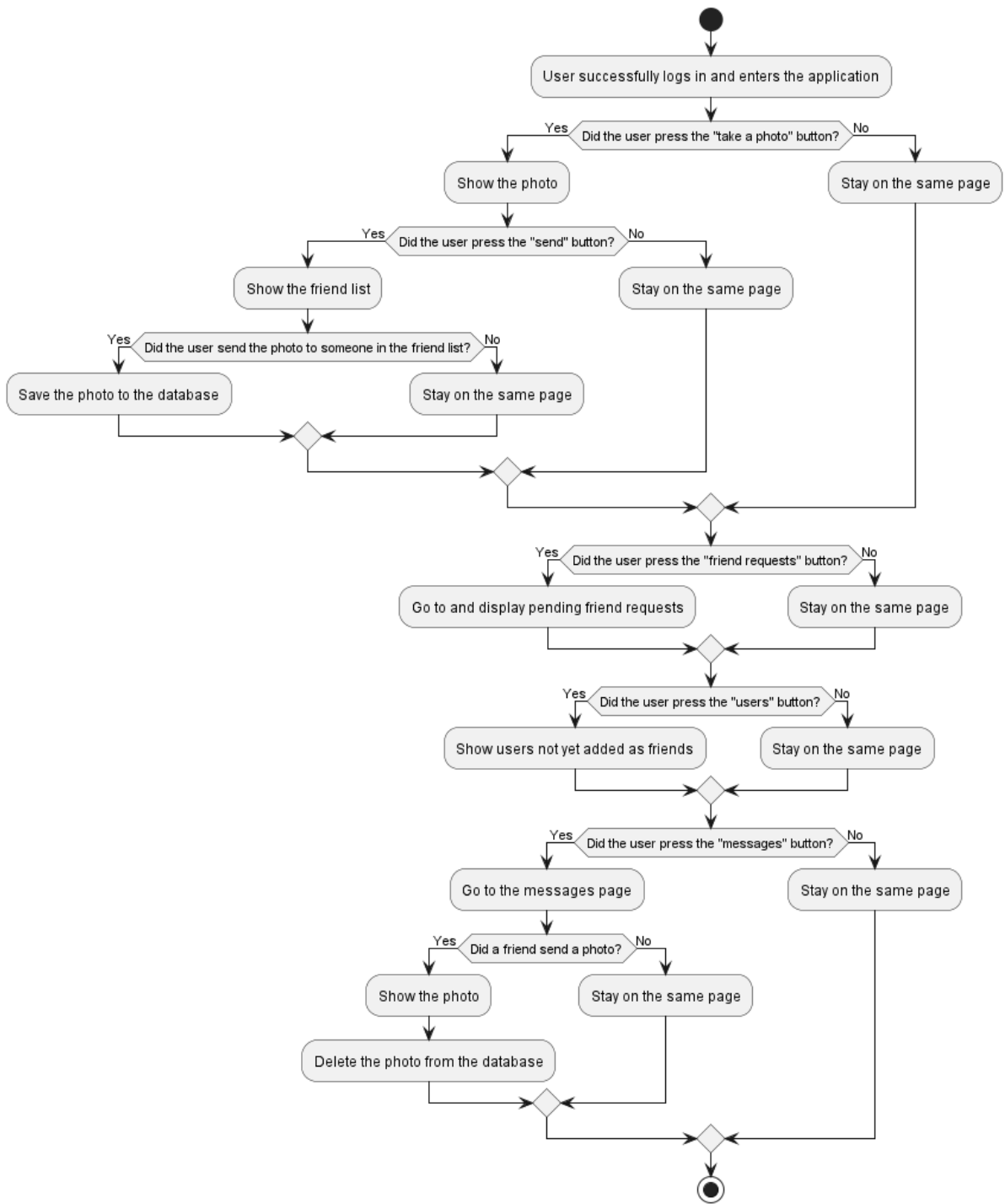
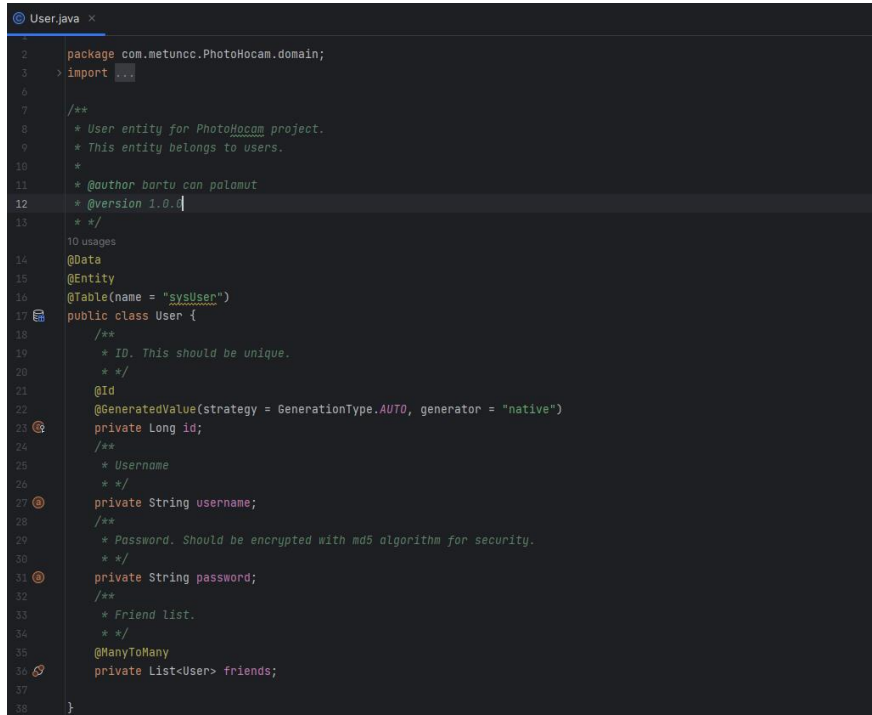


Figure 2 Activity Diagram

Figure 2 shows the algorithm of the rest of the project.

2.3 Parts of the project

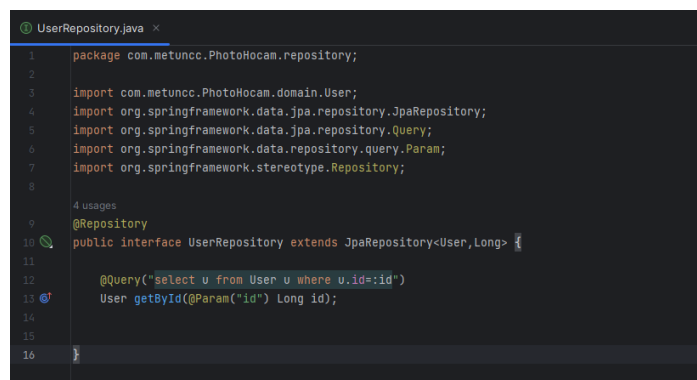
2.3.1 Back-end side



```
1 package com.metuncc.PhotoHocam.domain;
2
3 > import ...
4
5
6
7 /**
8  * User entity for PhotoHocam project.
9  * This entity belongs to users.
10  *
11  * @author bartu can palamut
12  * @version 1.0.1
13  */
14
15 10 usages
16 @Data
17 @Entity
18 @Table(name = "sysUser")
19 public class User {
20     /**
21     * ID. This should be unique.
22     */
23     @Id
24     @GeneratedValue(strategy = GenerationType.AUTO, generator = "native")
25     private Long id;
26     /**
27     * Username
28     */
29     private String username;
30     /**
31     * Password. Should be encrypted with md5 algorithm for security.
32     */
33     private String password;
34     /**
35     * Friend list.
36     */
37     @ManyToMany
38     private List<User> friends;
39 }
```

Figure 3 User Entity

As you can see from Figure 3, on our user entity, there are “id”, “username”, “password” and “friends” attributes. We encrypt user’s password MD5 algorithm before saving to database for security. Also, as you can see above, there is an attribute called “friends”. This attribute holds a list of users whose friend of this user.



```
1 package com.metuncc.PhotoHocam.repository;
2
3 import com.metuncc.PhotoHocam.domain.User;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.data.jpa.repository.Query;
6 import org.springframework.data.repository.query.Param;
7 import org.springframework.stereotype.Repository;
8
9 4 usages
10 @Repository
11 public interface UserRepository extends JpaRepository<User, Long> {
12     @Query("select u from User u where u.id=:id")
13     User getById(@Param("id") Long id);
14
15
16 }
```

Figure 4 User Repository

After that I created a simple repository for this class, and I created a very simple SQL command that takes a specific user from the database by id.

According to the Spring Security Documentation, we need to prepare some implementations for our project for creating security rules.

```
@JwtUserDetails.java
13
14 /**
15  * User Details Class implemented here.
16  *
17  * @author bartu can palamut
18  * @version 1.0.0
19  */
20 @Getter
21 @Setter
22 @AllArgsConstructor
23 public class JwtUserDetails implements UserDetails {
24     /**ID of User**/
25     private Long id;
26     /**Username of User**/
27     private String username;
28     /**Password of User**/
29     private String password;
30     /**Permissions of User**/
31     private Collection<? extends GrantedAuthority> authorities;
32
33     //Not implemented for now
34     @Override
35     public boolean isAccountNonExpired() { return true; }
36
37     //Not implemented for now
38     @Override
39     public boolean isAccountNonLocked() { return true; }
40
41     //Not implemented for now
42     @Override
43     public boolean isCredentialsNonExpired() { return true; }
44
45     //Not implemented for now
46     @Override
47     public boolean isPasswordNonExpired() { return true; }
48
49     //Not implemented for now
50     @Override
51     public boolean isPasswordNonExpired() { return true; }
52     @Override
```

Figure 5 UserDetails

First of all, I implemented the “UserDetails” interface. This interface comes from Spring Security dependency. In this interface, there are some methods, but we don’t want to use these methods. I just want to make a simple security chain because on our system, there is no user-blocking, role based authentication. I created the “JwtUserDetails” class and I added some attributes on this class, such as id of the user, username of the user, password of the user and permission of the user. We are not planning to use role-based authentication but based on documentation, I have to implement the “getAuthorities” method, so that we add this attribute here too. I set the default role to everyone in the future. After that, I created some classes for generating, validating and filtering authentication based on Json Web Token. As you can see below, based on Spring Security, there are some operations that I already mentioned above. These operations are necessary for a simple authentication process for Spring Boot.

```

12
13  /**
14   * That class generates JWT token for specific user with SECRET and expired time (in sec).
15   * */
16  4 usages  1 bartucank
17  @Component
18  public class JwtProvider {
19
20      @Value("${res.security.secret}")
21      private String SECRET;
22
23      @Value("${res.security.exp}")
24      private long EXP;
25
26  1 usage  1 bartucank
27  public String generateJwtToken(Authentication auth){
28      JwtUserDetails userDetails = (JwtUserDetails) auth.getPrincipal();
29      Date expDate= new Date(new Date().getTime() + EXP);
30      return Jwts.builder()
31          .setSubject(Long.toString(userDetails.getId()))
32          .setIssuedAt(new Date())
33          .setExpiration(expDate)
34          .signWith(SignatureAlgorithm.HS512,SECRET)
35          .compact();
36  }

```

Figure 6 JWT Provider / Generator

As you can see above, based on our configs which are secret key and expired second, json web token generates for user here.

```

14  1 bartucank
15  @Override
16  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
17      throws ServletException, IOException {
18      try {
19          String jwtToken = null;
20          String bearer = request.getHeader("Authorization");
21          if (StringUtils.hasText(bearer) && bearer.startsWith("Bearer ")){
22              jwtToken = bearer.substring(StringUtils.indexOf(bearer, "Bearer ").length() + 1);
23          }
24          Boolean isValidToken = false;
25          try {
26              Jwts
27                  .parser()
28                  .setSigningKey(SECRET)
29                  .parseClaimsJws(jwtToken);
30              isValidToken = true;
31          } catch (Exception e) {
32              isValidToken = false;
33          }
34          if (StringUtils.hasText(jwtToken) && isValidToken) {
35              Long id = Long.parseLong(
36                  Jwts
37                      .parser()
38                      .setSigningKey(SECRET)
39                      .parseClaimsJws(jwtToken)
40                      .getBody().getSubject()
41              );
42              User user = userRepository.findById(id);
43              UserDetails userDetails = user.toJwtUserDetails();
44              if (Objects.nonNull(userDetails)) {
45                  UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(user, null, userDetails.getAuthorities());
46                  auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
47                  SecurityContextHolder.getContext().setAuthentication(auth);
48              }
49          }
50      } catch (Exception e) {
51          return;
52      }
53      filterChain.doFilter(request, response);

```

Figure 7 Authentication Filtering

As you can see above, every request that is sent to our application is filtering here. Based on Spring Security documentation, I parsed JWT here then validated that token. Also, I matched user that are on JWT, with our database. If any error happens here, we return without setting Security Context Holder with Authentication, so that we ignore that requests.

```

1 package com.metuncc.PhotoHocam.security;
2
3 > import
4
5
6
7
8
9
10
11 /**
12  * This class represents an Authentication Entry Point for JWT authentication.
13  * Actually AuthenticationEntryPoint interface implemented here.
14  * If there is an unauthorized attempt, our service return 401 http status with is UNAUTHORIZED.
15  * @see org.springframework.security.web.AuthenticationEntryPoint
16  * */
17
18 @Component
19 public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {
20
21     @Override
22     public void commence(HttpServletRequest request, HttpServletResponse response,
23         AuthenticationException authException) throws IOException {
24         response.sendError(HttpServletResponse.SC_UNAUTHORIZED, authException.getMessage());
25     }
26 }

```

Figure 8 Authentication Entry Point

As you can see above, based on Spring Security documentation, we override commence method. If any request that is unauthorized, we return HTTP Status 401.

```

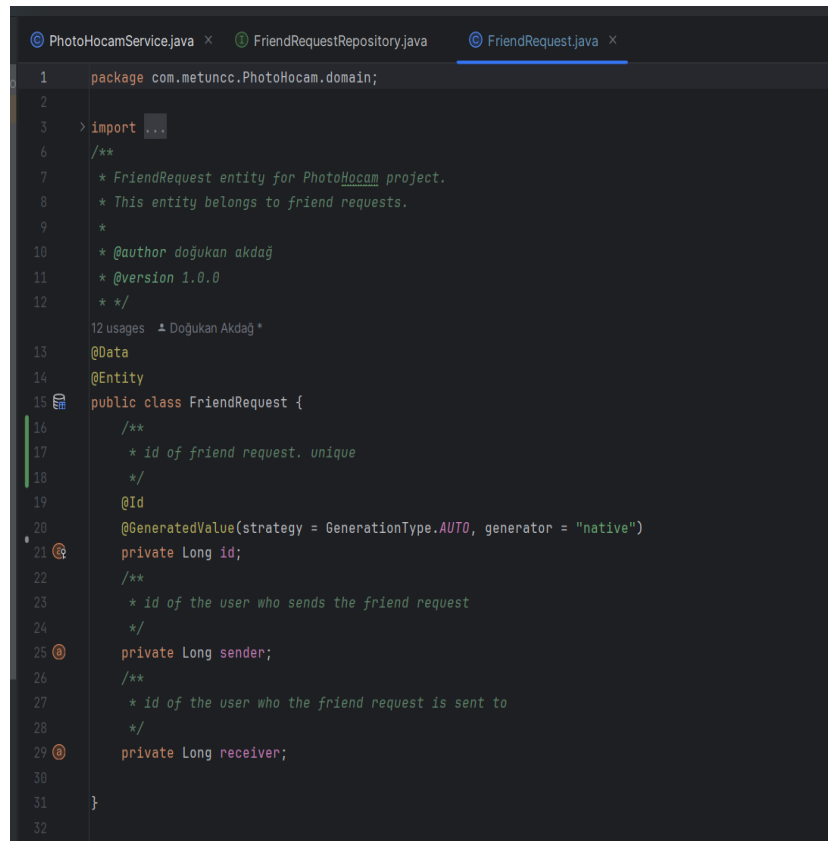
47
48
49 @Bean
50 public CorsFilter corsFilter() {
51     UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
52     CorsConfiguration config = new CorsConfiguration();
53     config.setAllowCredentials(false);
54     config.addAllowedOrigin("*");
55     config.addAllowedHeader("*");
56     config.addAllowedMethod("OPTIONS");
57     config.addAllowedMethod("HEAD");
58     config.addAllowedMethod("GET");
59     config.addAllowedMethod("PUT");
60     config.addAllowedMethod("POST");
61     config.addAllowedMethod("DELETE");
62     config.addAllowedMethod("PATCH");
63     source.registerCorsConfiguration(pattern: "**", config);
64     return new CorsFilter(source);
65 }
66
67
68 @Bean
69 public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {
70     httpSecurity
71         .cors().configure<HttpSecurity>
72         .and() .httpSecurity
73         .csrf().disable()
74         .exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and()
75         .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
76         .authorizeRequests().expressionInterceptUrlRegistry
77         .antMatchers("/api/login").permitAll()
78         .antMatchers("/api/register").permitAll()
79         .antMatchers("/api/**").hasAuthority("user")
80         .anyRequest().authenticated();
81     httpSecurity.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);
82     return httpSecurity.build();
83 }
84
85

```

Figure 9 Security Config

Then finally I created a SecurityConfig class that actually manages requests sent by clients to our system. As you can see below, I added necessary cors configs as allowed. For example, I added methods “PUT, POST, DELETE, PATCH, GET, OPTINS, HEAD” as allowed methods. Also, I set allowed origin and header as “*”. We are not planning to set a specific

origin because we do not know whether Google will give us a dynamic url. Then I created a “filterChain” method that enables the cors configurer, enabling jwtAuthenticationEntryPoint that I created already. Also I disabled csrf because we do not have any SSL certificate. Then I set some endpoints to enable without any token that are login and register. Rest of the all endpoints need jwt token.



```
1 package com.metuncc.PhotoHocam.domain;
2
3 > import ...
4
5 /**
6  * FriendRequest entity for PhotoHocam project.
7  * This entity belongs to friend requests.
8  *
9  * @author doğukan akdağ
10  * @version 1.0.0
11  * */
12 12 usages 1 Doğukan Akdağ *
13 @Data
14 @Entity
15 public class FriendRequest {
16     /**
17      * id of friend request. unique
18      */
19     @Id
20     @GeneratedValue(strategy = GenerationType.AUTO, generator = "native")
21     private Long id;
22     /**
23      * id of the user who sends the friend request
24      */
25     @
26     private Long sender;
27     /**
28      * id of the user who the friend request is sent to
29      */
30     @
31     private Long receiver;
32 }
```

Figure 10 FriendRequest Entity

I have created a FriendRequest entity that holds an unique id, id of the user who sends the friend request and also id of the user who the friend request is sent to.

```

1 package com.metuncc.PhotoHocam.repository;
2
3 > import
10
11 @Repository
12 public interface FriendRequestRepository extends JpaRepository<FriendRequest, Long>{
13     @Query("select u from FriendRequest u where u.id=:id")
14     FriendRequest findById(@Param("id") Long id);
15
16     @Query("select u from FriendRequest u where u.receiver=:receiver")
17     List<FriendRequest> findByReceiver(@Param("receiver")Long receiver);
18
19     @Query("select u from FriendRequest u where u.sender=:sender")
20     List<FriendRequest> findBySender(@Param("sender")String sender);
21 }
22
23

```

Figure 11 FriendRequest Repository

I have created a FriendRequest repository for the class. Then I have created simple SQL queries to find friend requests by the id of the friend request, id of the sender and id of the receiver.

```

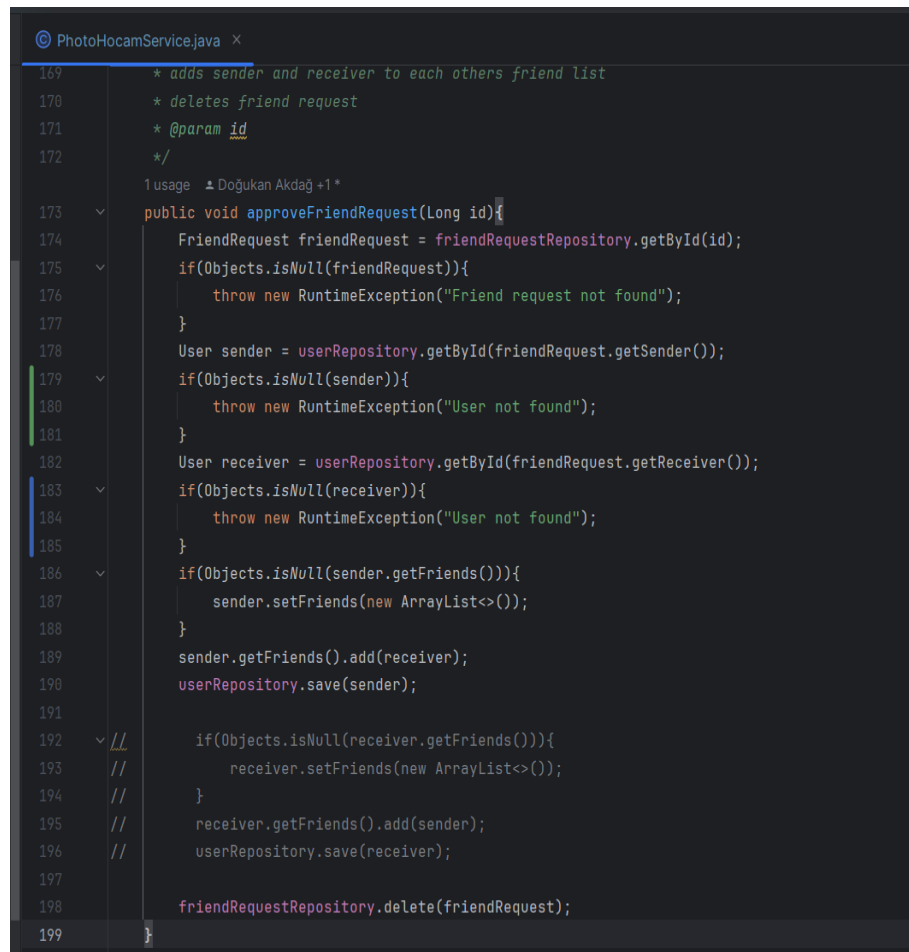
157
158 /**
159  * Creates a friend request
160  * gets receiver id as parameter
161  * @param receiver
162  */
163 1 usage  Doğukan Akdağ
164 public void sendFriendRequest(Long receiver){
165     if(Objects.isNull(userRepository.getById(getCurrentUserId()))){
166         throw new RuntimeException("User not found");
167     }
168     if(Objects.isNull(userRepository.getById(receiver))){
169         throw new RuntimeException("Receiver not found");
170     }
171     FriendRequest friendRequest = new FriendRequest();
172     friendRequest.setSender(getCurrentUserId());
173     friendRequest.setReceiver(receiver);
174     friendRequestRepository.save(friendRequest);
175     return;
176 }

```

Figure 12 sendFriendRequest service

I have created a service for sending friend requests. It takes the id of the receiver as a parameter. Id of the sender is acquired using the getCurrentUserId service that Bartu wrote which gets the user id from the authentication. After checking if the sender and receiver users

exist in the repository a friend request gets created and saved to the repository. Otherwise an exception is thrown.



```
169  * adds sender and receiver to each others friend list
170  * deletes friend request
171  * @param id
172  */
173  1 usage  1 Doğukan Akdağ +1 *
174  public void approveFriendRequest(Long id){
175      FriendRequest friendRequest = friendRequestRepository.getById(id);
176      if(Objects.isNull(friendRequest)){
177          throw new RuntimeException("Friend request not found");
178      }
179      User sender = userRepository.getById(friendRequest.getSender());
180      if(Objects.isNull(sender)){
181          throw new RuntimeException("User not found");
182      }
183      User receiver = userRepository.getById(friendRequest.getReceiver());
184      if(Objects.isNull(receiver)){
185          throw new RuntimeException("User not found");
186      }
187      if(Objects.isNull(sender.getFriends())){
188          sender.setFriends(new ArrayList<>());
189      }
190      sender.getFriends().add(receiver);
191      userRepository.save(sender);
192      // if(Objects.isNull(receiver.getFriends())){
193      //     receiver.setFriends(new ArrayList<>());
194      // }
195      // receiver.getFriends().add(sender);
196      // userRepository.save(receiver);
197
198      friendRequestRepository.delete(friendRequest);
199  }
```

Figure 13 approveFriendRequest service

Then I finally created a service that approves friend requests. It takes the id of the friend request as a parameter. First I check if the friend request with the parameter id exists in the repository. Then I check if the sender and receiver exist in the user repository and throw an exception if they don't. After pulling the users from the user repository using the sender and receiver ids, if the user's friend lists are empty it creates a new empty list for them. Then it adds both users to each other's friend list. Finally it deletes the friend request from the repository. But after testing we encountered a bug that will put our system in a loop which we explained below in week 11. To prevent it we removed the other half of our adding friend operations as you can see in the commented out lines.

```

* @author Nurberat Gökmén
* @version 1.0.0
*/
13 usages 1 nurgokmen *
@Entity
@Data
public class Image {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "native")
    /**
     * unique id for each image
     */
    private Long id;

    /**
     * user who sends the image
     * Every user can send an image to more than one receiver
     */
    @ManyToOne
    private User sender;

    /**
     * user who receives the image
     * each user can receive an image from more than one friend
     */
    @ManyToOne
    private User receiver;

    @Lob
    @Type(type = "org.hibernate.type.ImageType")
    /**
     * to store the binary data of the image. for large objects
     */
    private byte[] data;
}

```

Figure 14 Image Entity

I have created an Image entity. Image entity holds a private and unique id. A user type receiver and sender. Also, I used @Lob in order to handle large sizes of data. In the first weeks I used receiver and sender as Long type because we haven't yet opened the git repository so I wasn't able to access the User entity Bartu has created. But later on after opening the git repository, and merged everyone's work, so I changed the receiver and sender type from long to User.

```

3 usages  nurgokmen
@Repository
public interface ImageRepository extends JpaRepository<Image,Long> {

    nurgokmen
    @Query("select i from Image i where i.id=:id")
    Image getById(@Param("id") Long id);

    1 usage  nurgokmen
    @Query("select i from Image i where i.receiver.id=:user")
    List<Image> getImages(@Param("user") Long user);

```

Figure 15 Image Repository

I have created an Image repository which extends the JpaRepository. In this repository, I included two SQL queries: One query is for getting a single image from its unique id and the second query is for obtaining the list of images sent to a specific user. The list of images is retrieved by its receiver's id.


```

    * @return true if sent operation is successful, false if receiver not found
    */
    1 usage  1 nurgokmen *
    public boolean sendImage(MultipartFile file, Long userID) {

        User receiver = userRepository.getById(userID);
        User sender = userRepository.getById(getCurrentUserId());

        if (receiver == null) {
            return false;
        }
        try {
            Image img = new Image();
            var deflater = new Deflater();
            deflater.setLevel(Deflater.BEST_COMPRESSION);
            deflater.setInput(file.getBytes());
            deflater.finish();

            ByteArrayOutputStream outputStream = new ByteArrayOutputStream(file.getBytes().length);
            byte[] tmp = new byte[4*1024];
            while (!deflater.finished()) {
                int size = deflater.deflate(tmp);
                outputStream.write(tmp, 0, size);}
            try {
                outputStream.close();
            } catch (Exception e) {
            }
            img.setData( outputStream.toByteArray());
            img.setReceiver(receiver);
            img.setSender(sender);
            imageRepository.save(img);
        } catch (IOException e) {
            return false;
        }

        return true;    }

```

Figure 16 Send Image Service

I have created a sendImage service which takes a multipartFile which is an image in our case and a long id, which is the unique id of the receiver. This method handles the sending of an image operation. The method checks the sender and the receiver by getting their ids from the user repository. If the id given as parameter in this method is not matching with any of the ids in the user repository, the operation can not proceed. Then, I created an Image object to represent the file in operation. I used deflater in order to compress the file. After compression the image bytes are written into the ByteArrayOutputStream. Later on, I sent the image data, receiver information and sender information to the Image repository. I used try catch block so if any exception occurs during this process, the method will return false.

```

/**
 * deletes the image specified by its ID
 * @param id
 */

no usages  🧑 nurgokmen
public void deleteImage(Long id){

    imageRepository.deleteById(id);

}

```

Figure 17 Delete Image Service

I created a delete Image method in order to delete the images by their ids. This method takes a long id which is the id of the image as a parameter.

```

/**
 * this class is used for transferring the data of the image within the application
 */

8 usages  🧑 nurgokmen
@Data
public class Imagedto {

    /**
     * username of the user sent the image
     */
    private String username;

    /**
     * list of the images sent from the specified user
     */
    private List<Image> imageList;
}

```

Figure 18 Imagedto Class

This java class I implemented is a data transfer object class. It contains a username and a list of images. It contains the images sent by the specified username and the usernames which sent the images.

```
/**
 * gets a list of the image objects sent by a user. Based on the sender's username.
 * @return list of Imagedto, containing the username of the sender and the images they sent
 */
1 usage  1 nurgokmen
public List<Imagedto> getImageList(){

    Long me = getCurrentUserId();
    List<Image> imgs = imageRepository.getImages(me);
    List<Imagedto> result = new ArrayList<>();
    for (Image img : imgs) {

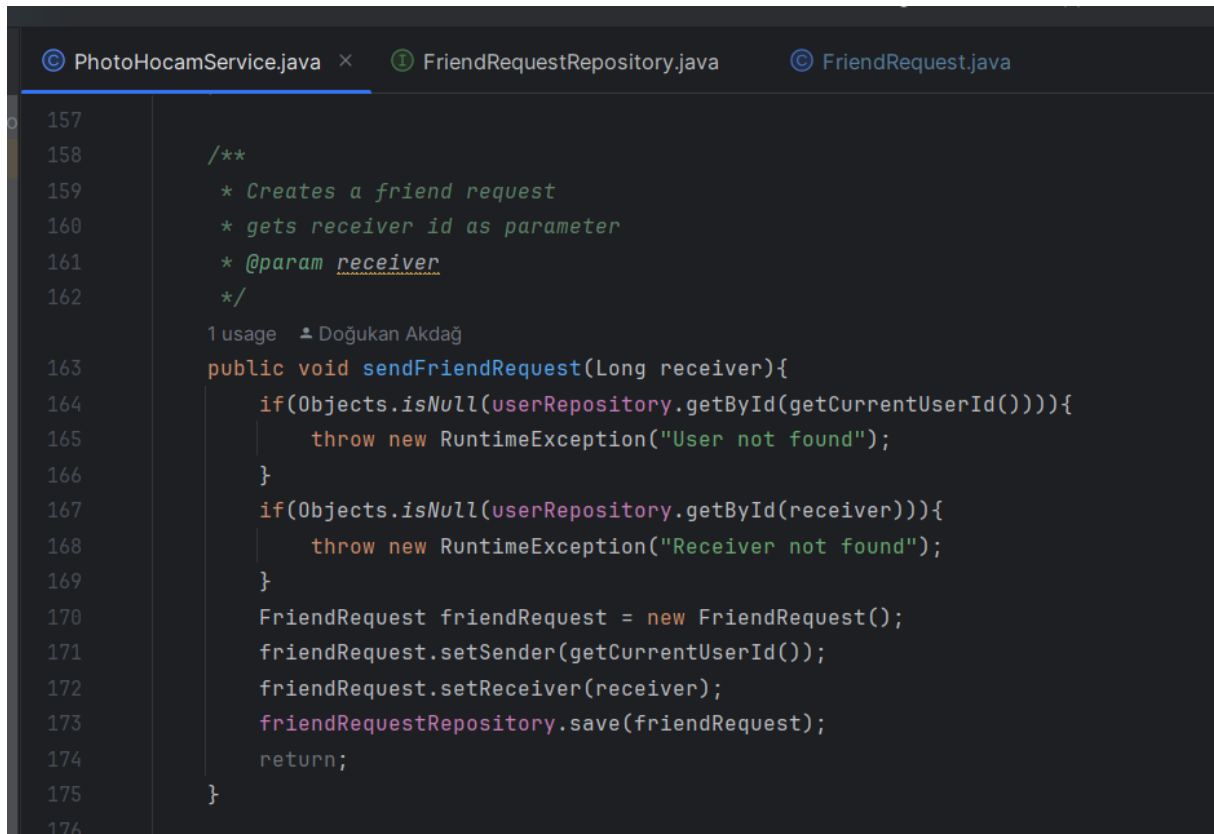
        Boolean check = false;
        for (Imagedto imagedto : result) {
            if(imagedto.getUsername().equals(img.getSender().getUsername())){
                check = true;
                imagedto.getImageList().add(img);
            }
        }
        if(!check){
            Imagedto imagedto = new Imagedto();
            imagedto.setUsername(img.getSender().getUsername());
            imagedto.setImageList(new ArrayList<>());
            imagedto.getImageList().add(img);
        }
    }

    return result;
}
```

Figure 19 Get Image List Service

I have created a getImageList service. This method is used to get the list of images sent by the currently logged in user. It returns the list of images from the Image Repository sent by the current user's username. This method iterates through the images and adds the image found to the list checking by the sender's username. If the sender's username is included in the list then the method adds the current image to the Imagedto list. If the sender's username is not

included in the list then the method creates a new `Imagedto` for that sender.. The method returns a list of `Imagedto` objects, objects contain the list of images and the username of the sender for that images.

A screenshot of an IDE window showing three tabs: 'PhotoHocamService.java', 'FriendRequestRepository.java', and 'FriendRequest.java'. The 'PhotoHocamService.java' tab is active, displaying a Java method named 'sendFriendRequest'. The method takes a 'Long receiver' parameter. It includes two null checks: one for the current user (throwing 'User not found') and one for the receiver (throwing 'Receiver not found'). If both checks pass, it creates a 'FriendRequest' object, sets the sender to the current user ID and the receiver to the provided ID, and saves it to the 'friendRequestRepository'. The method returns nothing. Line numbers 157 through 176 are visible on the left margin.

```
157
158     /**
159      * Creates a friend request
160      * gets receiver id as parameter
161      * @param receiver
162      */
163     public void sendFriendRequest(Long receiver){
164         if(Objects.isNull(userRepository.getById(getCurrentUserId()))){
165             throw new RuntimeException("User not found");
166         }
167         if(Objects.isNull(userRepository.getById(receiver))){
168             throw new RuntimeException("Receiver not found");
169         }
170         FriendRequest friendRequest = new FriendRequest();
171         friendRequest.setSender(getCurrentUserId());
172         friendRequest.setReceiver(receiver);
173         friendRequestRepository.save(friendRequest);
174         return;
175     }
176
```

Figure 20 FriendRequest Repository

I have created a service for sending friend requests. It takes the id of the receiver as a parameter. Id of the sender is acquired using the `getCurrentUserId` service that Bartu wrote which gets the user id from the authentication. After checking if the sender and receiver users exist in the repository a friend request gets created and saved to the repository. Otherwise an exception is thrown.

```

169      * adds sender and receiver to each others friend list
170      * deletes friend request
171      * @param id
172      */
173      public void approveFriendRequest(Long id){
174          FriendRequest friendRequest = friendRequestRepository.getById(id);
175          if(Objects.isNull(friendRequest)){
176              throw new RuntimeException("Friend request not found");
177          }
178          User sender = userRepository.getById(friendRequest.getSender());
179          if(Objects.isNull(sender)){
180              throw new RuntimeException("User not found");
181          }
182          User receiver = userRepository.getById(friendRequest.getReceiver());
183          if(Objects.isNull(receiver)){
184              throw new RuntimeException("User not found");
185          }
186          if(Objects.isNull(sender.getFriends())){
187              sender.setFriends(new ArrayList<>());
188          }
189          sender.getFriends().add(receiver);
190          userRepository.save(sender);
191
192          // if(Objects.isNull(receiver.getFriends())){
193          //     receiver.setFriends(new ArrayList<>());
194          // }
195          // receiver.getFriends().add(sender);
196          // userRepository.save(receiver);
197
198          friendRequestRepository.delete(friendRequest);
199      }

```

Figure 21 approveFriendRequest service

Then I finally created a service that approves friend requests. It takes the id of the friend request as a parameter. First I check if the friend request with the parameter id exists in the repository. Then I check if the sender and receiver exist in the user repository and throw an exception if they don't. After pulling the users from the user repository using the sender and receiver ids, if the user's friend lists are empty it creates a new empty list for them. Then it adds both users to each other's friend list. Finally it deletes the friend request from the repository. But after testing we encountered a bug that will put our system in a loop which we explained below in week 11. To prevent it we removed the other half of our adding friend operations as you can see in the commented out lines.

```

* @author Nurberat Gökmen
* @version 1.0.0
*/
13 usages  nurgokmen *
@Entity
@Data
public class Image {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "native")
    /**
     * unique id for each image
     */
    private Long id;

    /**
     * user who sends the image
     * Every user can send an image to more than one receiver
     */
    @ManyToOne
    private User sender;

    /**
     * user who receives the image
     * each user can receive an image from more than one friend
     */
    @ManyToOne
    private User receiver;

    @Lob
    @Type(type = "org.hibernate.type.ImageType")
    /**
     * to store the binary data of the image. for large objects
     */
    private byte[] data;
}

```

Figure 22 Image Entity

I have created an Image entity. Image entity holds a private and unique id. A user type receiver and sender. Also, I used @Lob in order to handle large sizes of data. In the first weeks I used receiver and sender as Long type because we haven't yet opened the git repository so I wasn't able to access the User entity Bartu has created. But later on after

opening the git repository, and merged everyone's work, so I changed the receiver and sender type from long to User.

```

    * @return true if sent operation is successful, false if receiver not found
    */
    1 usage  1 nurgokmen *
    public boolean sendImage(MultipartFile file, Long userID) {

        User receiver = userRepository.getById(userID);
        User sender = userRepository.getById(getCurrentUserId());

        if (receiver == null) {
            return false;
        }
        try {
            Image img = new Image();
            var deflater = new Deflater();
            deflater.setLevel(Deflater.BEST_COMPRESSION);
            deflater.setInput(file.getBytes());
            deflater.finish();
            ByteArrayOutputStream outputStream = new ByteArrayOutputStream(file.getBytes().length);
            byte[] tmp = new byte[4*1024];
            while (!deflater.finished()) {
                int size = deflater.deflate(tmp);
                outputStream.write(tmp, off: 0, size);
            }
            try {
                outputStream.close();
            } catch (Exception e) {
            }
            img.setData( outputStream.toByteArray());
            img.setReceiver(receiver);
            img.setSender(sender);
            imageRepository.save(img);
        } catch (IOException e) {
            return false;
        }
        return true;
    }

```

Figure 23 Send Image Service

I have created a sendImage service which takes a multipartFile which is an image in our case and a long id, which is the unique id of the receiver. This method handles the sending of an image operation. The method checks the sender and the receiver by getting their ids from the user repository. If the id given as parameter in this method is not matching with any of the ids in the user repository, the operation can not proceed. Then, I created an Image object to represent the file in operation. I used deflater in order to compress the file. After compression the image bytes are written into the ByteArrayOutputStream. Later on, I sent the image data, receiver information and sender information to the Image repository. I used try catch block so if any exception occurs during this process, the method will return false.

```

/**
 * deletes the image specified by its ID
 * @param id
 */

no usages  🧑 nurgokmen
public void deleteImage(Long id){

    imageRepository.deleteById(id);

}

```

Figure 24 Delete Image Service

I created a delete Image method in order to delete the images by their ids. This method takes a long id which is the id of the image as a parameter.

```

/**
 * this class is used for transferring the data of the image within the application
 */

8 usages  🧑 nurgokmen
@Data
public class Imagedto {

    /**
     * username of the user sent the image
     */
    private String username;

    /**
     * list of the images sent from the specified user
     */
    private List<Image> imageList;
}

```

Figure 25 Imagedto Class

This java class I implemented is a data transfer object class. It contains a username and a list of images. It contains the images sent by the specified username and the usernames which sent the images.

```
/**
 * gets a list of the image objects sent by a user. Based on the sender's username.
 * @return list of Imagedto, containing the username of the sender and the images they sent
 */
1 usage  1 nurgokmen
public List<Imagedto> getImageList(){

    Long me = getCurrentUserId();
    List<Image> imgs = imageRepository.getImages(me);
    List<Imagedto> result = new ArrayList<>();
    for (Image img : imgs) {

        Boolean check = false;
        for (Imagedto imagedto : result) {
            if(imagedto.getUsername().equals(img.getSender().getUsername())){
                check = true;
                imagedto.getImageList().add(img);
            }
        }
        if(!check){
            Imagedto imagedto = new Imagedto();
            imagedto.setUsername(img.getSender().getUsername());
            imagedto.setImageList(new ArrayList<>());
            imagedto.getImageList().add(img);
        }
    }

    return result;
}
```

Figure 26 Get Image List Service

I have created a getImageList service. This method is used to get the list of images sent by the currently logged in user. It returns the list of images from the Image Repository sent by the current user's username. This method iterates through the images and adds the image found to the list checking by the sender's username. If the sender's username is included in the list then

the method adds the current image to the Imagedto list. If the sender's username is not included in the list then the method creates a new Imagdto for that sender.. The method returns a list of Imagedto objects, objects contain the list of images and the username of the sender for that images.

2.3.2 Front-end side

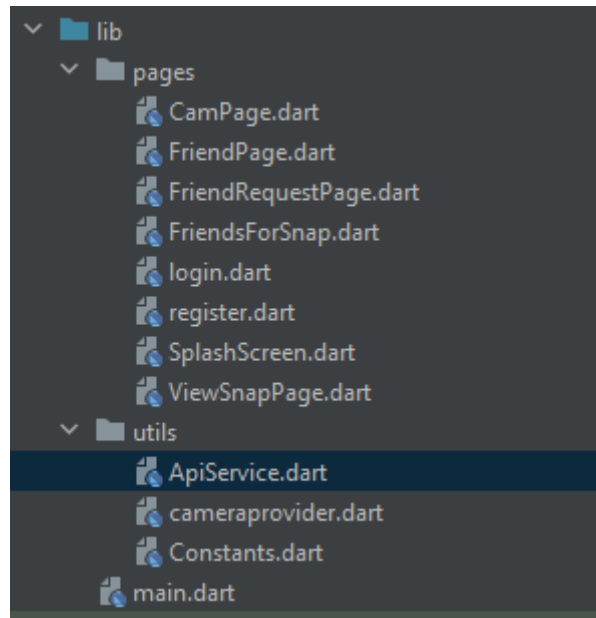


Figure 27 Project Structur

Our project structure look like this. On pages package, we created some pages that will be shown to user. On utils package, we created some classes for convenience and to perform some operations over and over again.

```
class Constants {  
  // static const String apiBaseUrl = "http://cloud495-407311.oa.r.appspot.com";  
  static const String apiBaseUrl = "https://a732-78-135-49-15.ngrok-free.app";  
}
```

Figure 28 Constants Class

As you can see above, we hold some constants here. For example, we hold end point of the back-end server here.

```

3 import 'package:camera/camera.dart';
4 import 'package:flutter/foundation.dart';
5 import 'package:flutter_cache_manager/flutter_cache_manager.dart';
6 class CameraProvider extends ChangeNotifier{
7     late CameraController cameraController;
8     late List<CameraDescription> cameraList;
9
10    void init() async {
11        cameraList = await availableCameras();
12        cameraController = CameraController(cameraList.first, ResolutionPreset.ultraHigh,
13            enableAudio: true);
14        cameraController.initialize().then((_) {
15            notifyListeners();
16        });
17        notifyListeners();
18    }
19
20    secondCamera() {
21        final currentCamera = cameraController.description.lensDirection;
22        for (var cam in cameraList) {
23            if (cam.lensDirection != currentCamera) {
24                cameraController = CameraController(cam, ResolutionPreset.ultraHigh,
25                    enableAudio: true);
26                cameraController.initialize().then((_) {
27                    notifyListeners();
28                });
29                break;
30            }
31        }
32        return;
33    }
34
35
36    CameraController getController() => cameraController;
37 }

```

Figure 29 Camera Provider

As you can see above, I created a class that called CameraProvider. Actually it extends change notifier, indicating its ability to notify listeners about changes in its state. I used camera library of the flutter for controlling camera.

```

import 'dart:convert';
import 'dart:io';
import 'package:http/http.dart' as http;
import 'package:flutter_secure_storage/flutter_secure_storage.dart';
import 'Constants.dart';

class ApiService {
  final FlutterSecureStorage storage = FlutterSecureStorage();

  Future<String?> getJwtToken() async {
    try {
      final jwtToken = await storage.read(key: 'jwt_token');
      return jwtToken;
    } catch (error) {
      return "NOT_FOUND";
    }
  }

  Future<void> saveJwtToken(String token) async {
    await storage.write(key: 'jwt_token', value: token);
  }

  Future<void> logout() async {
    await storage.delete(key: 'jwt_token');
  }

  Future<bool> loginRequest(dynamic body) async {
    final response = await http.post(
      Uri.parse('${Constants.apiUrl}/api/login'),
      headers: {
        'Content-Type': 'application/json',
      },
      body: jsonEncode(body),
    );
    Map<String, dynamic> jsonResponse = jsonDecode(response.body);
    if (response.statusCode == 200) {
      saveJwtToken(jsonResponse['jwt']);
      return true;
    }
    return false;
  }
}

```

Figure 30 ApiService Class

As you can see above, our ApiService.dart file look like this. We hold JWT that generated by back-end side with Flutter Secure Storage library. When a user logged in, we called

“saveJwtToken” method that saves jwt to storage. Also, we will call “getJwtToken” method when we will call api requests. Also we implemented logout method that deletes jwt token from storage. As you can see, we implemented loginRequest method that takes body,generate json body from that body, and send it to the backend. If http status code is 200, our process is successful, so we save jwt token and return with value of “true”.

```
Future<bool> registerRequest(dynamic body) async {
  final response = await http.post(
    Uri.parse('${Constants.apiUrl}/api/register'),
    headers: {
      'Content-Type': 'application/json',
    },
    body: jsonEncode(body),
  );
  if (response.statusCode == 200) {
    return true;
  }
  return false;
}

Future<List<dynamic>> getUnfriends() async {
  final jwtToken = await getJwtToken();
  if (jwtToken != null) {
    final response = await http.get(
      Uri.parse('${Constants.apiUrl}/api/user/getunfriends'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $jwtToken',
      },
    );
    if (response.statusCode == 200) {
      final Map<String, dynamic> data = json.decode(response.body);
      return data['userDTOList'];
    } else {
      throw Exception('Failed to load data');
    }
  } else {
    throw Exception('JWT Token not found');
  }
}
```

Figure 31 Api Calls

As you can see above, we implemented registerRequest method that takes body,generate json body from that body, and send it to the backend. If http status code is 200, our process is successful, so we return with value of “true”. Also we implemented getUnfriends method that send api request to backend for getting users whose are not friend with caller user. If http status code is 200, our process is successful, so we generated map of string,dynamic data from response’s body and returned this value.

```

Future<List<dynamic>> getFriendRequestList() async {
  final jwtToken = await getJwtToken();
  if (jwtToken != null) {
    final response = await http.get(
      Uri.parse('${Constants.apiUrl}/api/user/getFriendRequestList'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $jwtToken',
      },
    );

    if (response.statusCode == 200) {
      final Map<String, dynamic> data = json.decode(response.body);
      print("ok");
      print(data);
      return data['userDTOList'];
    } else {
      throw Exception('Failed to load data');
    }
  } else {
    throw Exception('JWT Token not found');
  }
}

Future<List<dynamic>> getFriends() async {
  final jwtToken = await getJwtToken();
  if (jwtToken != null) {
    final response = await http.get(
      Uri.parse('${Constants.apiUrl}/api/user/getfriends'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $jwtToken',
      },
    );

    if (response.statusCode == 200) {
      final Map<String, dynamic> data = json.decode(response.body);
      return data['userDTOList'];
    } else {
      throw Exception('Failed to load data');
    }
  } else {
    throw Exception('JWT Token not found');
  }
}

```

Figure 32 Api Calls

As you can see above, we implemented getFriendRequestList and getFriends methods. The process is same as the previously described service.

```

Future<void> addFriend(int id) async {
  final jwtToken = await getJwtToken();
  if (jwtToken != null) {
    final response = await http.post(
      Uri.parse('${Constants.apiUrl}/api/user/addFriend?id=$id'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $jwtToken',
      },
    );
  }
}

Future<void> approveFriendRequest(int id) async {
  final jwtToken = await getJwtToken();
  if (jwtToken != null) {
    final response = await http.post(
      Uri.parse('${Constants.apiUrl}/api/user/approveFriendRequest?id=$id'),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer $jwtToken',
      },
    );
  }
}

```

Figure 33 Api Calls

As you can see above, we implemented addFriend and approveFriendRequest methods. Due to backend did not have message for these endpoints we did not return any value for that methods.

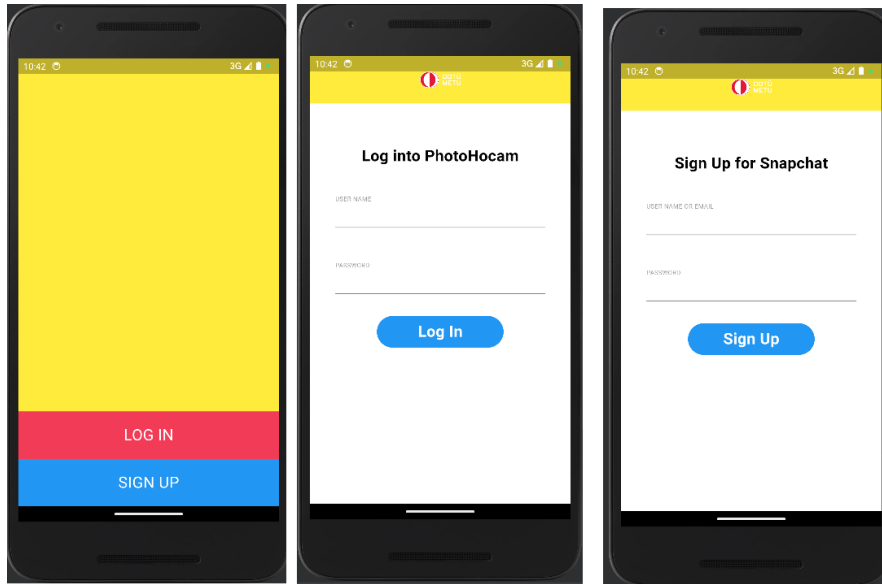


Figure 34 UI of Authentication

As you can see from the figure 34, there are authentication related pages of our client. The first photo of the figure is splash page. When the user opens our application, the user navigated to this page. Users can click log in or sign-up button. If the user clicks log in button, user navigated to the log in page. User can login with username and password from this page. When the user try to login to the our system, our back-end services generates secret JWT for authentication if username and password combination are correct. If the user clicks sign-up button, user navigated to the register page. User can register to our system with username and password from this page. After that user navigated to log-in page again.

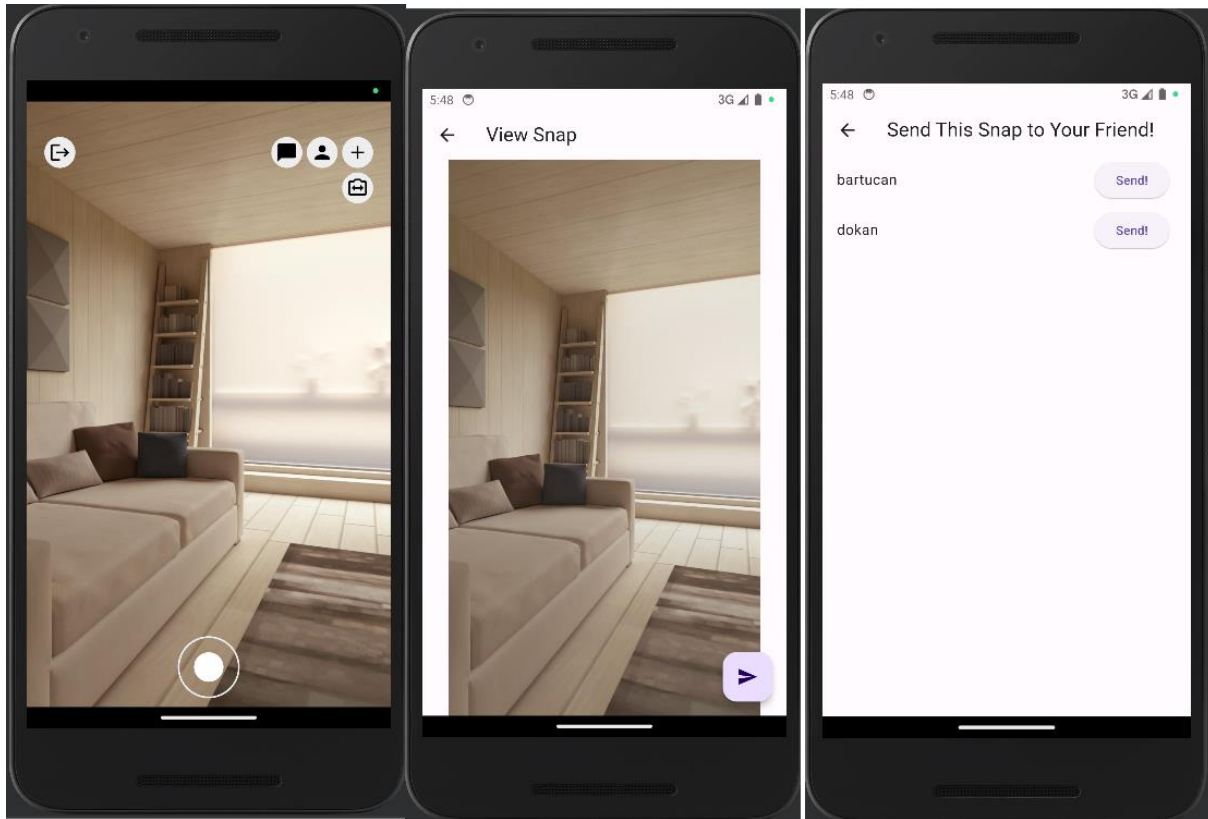


Figure 35 UI for Camera

As you can see from the figure 35, there are camera related pages. The first page is actually home page for our system. User can take a snap/photo with camera. Also user can rotate the camera. When user click take photo button, user navigated to View Snap page. User can check the photo from here. If user want to send this photo to his/her friend, s/he can click sent button. If the user click that button, the user navigated “Send This Snap to Your Friend!” page. On this page, user’s friend list will be listed. User can click send button to send snap/image.

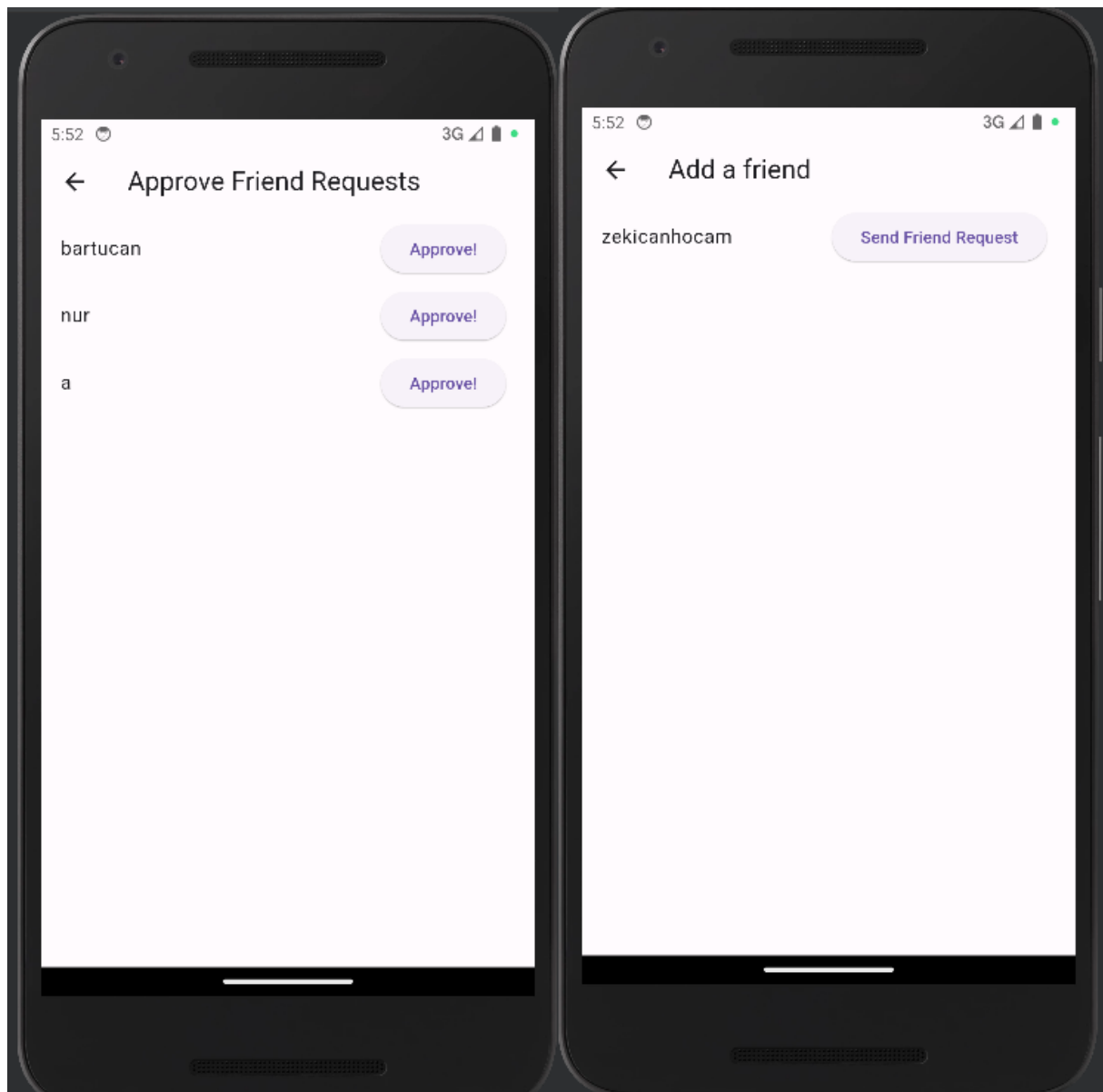


Figure 36 UI for Un/Friends

As you can see from the figure 36, “Approve Friend Requests” and “Add a friend” pages look like this. When this user receives a friend request from someone else, the request will be listed on the first page. User can approve it. When the user wants to send a friend request, the user can navigate to the second page. On the second page, users who are not friends of the user using the application and to whom a friend request has not been sent are listed.

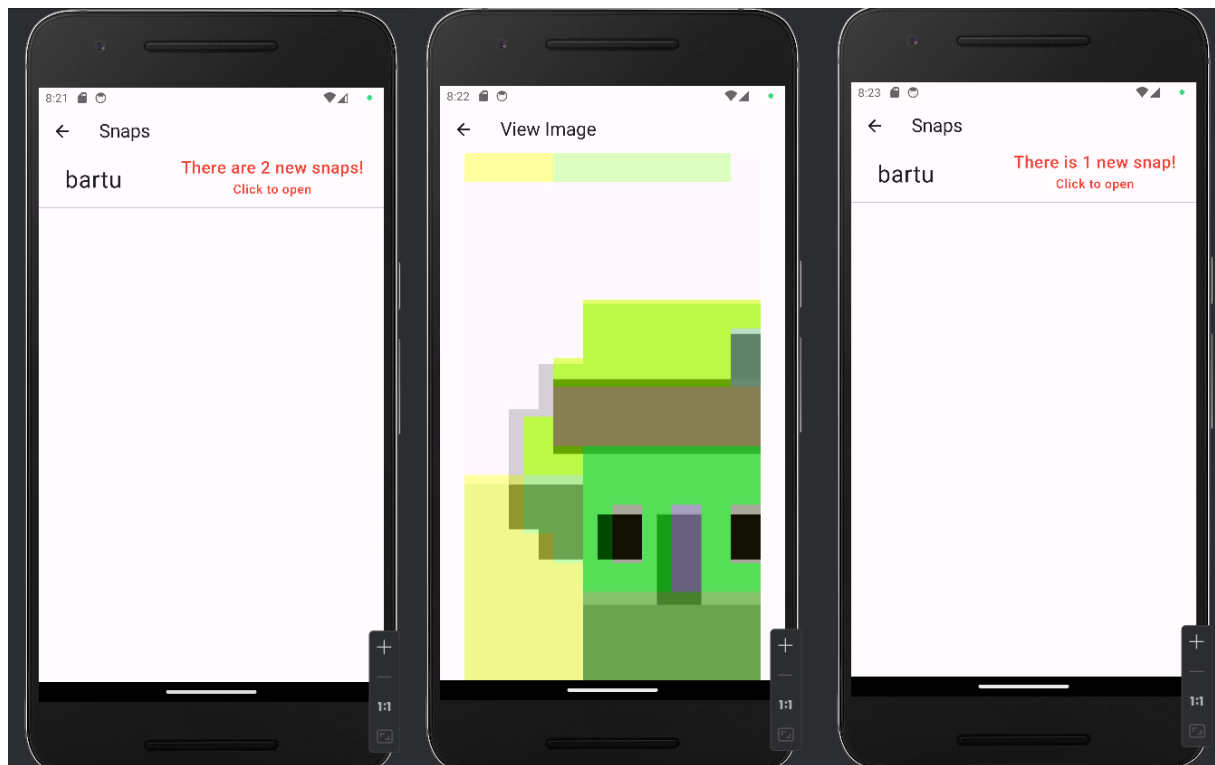


Figure 37 UI for Messages

We have implemented the Snaps page that will display to user any unopened messages they got from other users. When the page is first opened it will fetch all the messages user got from the database and display them with whom they got the message and how many snaps they got. When the photo is displayed after clicking anywhere in screen it will first delete the photo from the database and then go to the previous page and reinitialize the page, so our Snaps page is up to date.

```

Future<List<dynamic>> getImageList() async {
  final jwtToken = await getJwtToken();
  if (jwtToken != null) {
    try {
      final response = await http.get(
        Uri.parse('${Constants.apiUrl}/api/user/getImageList'),
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer $jwtToken',
        },
      );
      final Map<String, dynamic>? responseData = json.decode(response.body);
      return responseData!['list'];
    } catch (e) {
      throw Exception('');
    }
  } else {
    throw Exception('JWT Token not found');
  }
}

```

Figure 38 Api Call

Figure 38 shows our API Service for fetching the messages from database. We are sending a request to our /getImageList endpoint on our backend server and decode our response to map in front end.

```
Future<void> deletePhoto(int id) async {  
    final jwtToken = await getJwtToken();  
    if (jwtToken != null) {  
        final response = await http.delete(  
            Uri.parse('${Constants.apiUrl}/api/deletePhoto?id=$id'),  
            headers: {  
                'Content-Type': 'application/json',  
                'Authorization': 'Bearer $jwtToken',  
            },  
        );  
    }  
}
```

Figure 39 Api Call

Figure 39 shows our deletePhoto API service. We are sending image id to our deletePhoto endpoint in our backend server to delete the image from our database.

3. Statistics

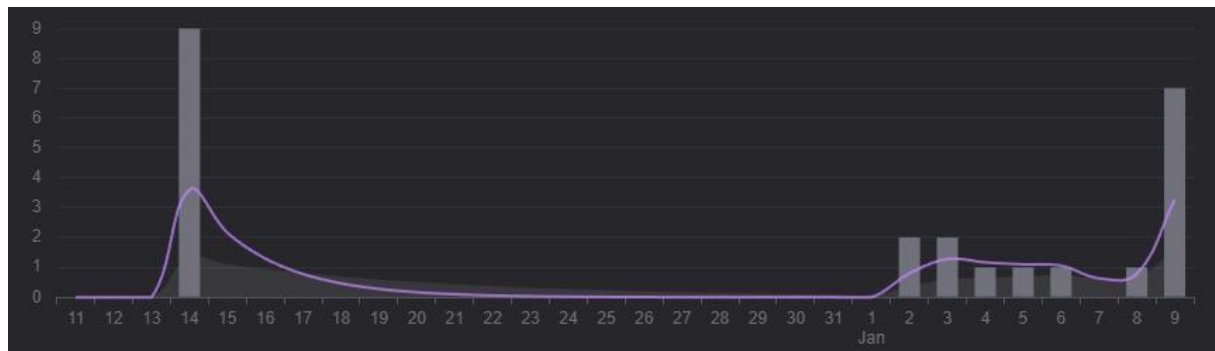


Figure 40 Commit History

Figure 40 shows the commit history of our project.

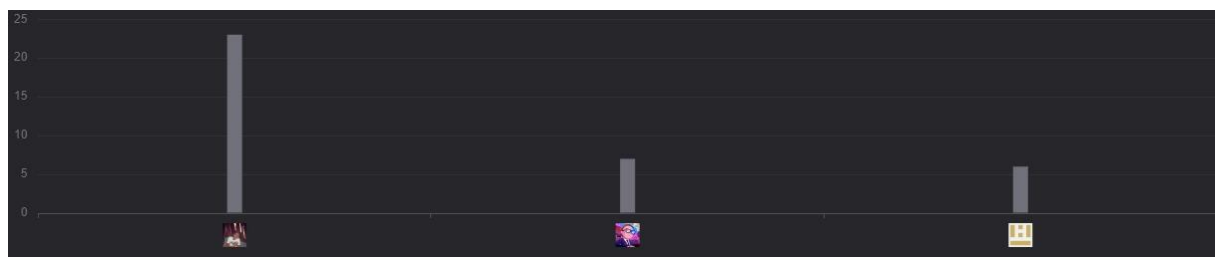


Figure 41 Individual Commits

Figure 41 shows the individual commit history of our project.

- The project was completed with 37 commits in total.
- The commit was made in 8 days in total.
- There are a total of 4,743 lines of code in the project.

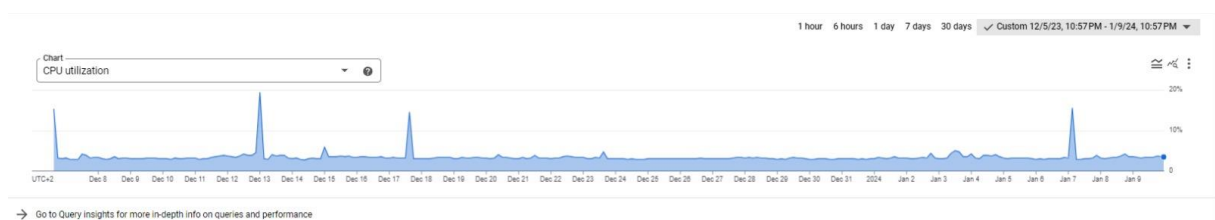


Figure 42 SQL Cloud CPU Utilization

Figure 42 shows SQL Cloud CPU Utilization for our project.



Figure 43 App Engine Memory Usage

Figure 43 shows memory usage of App Engine for 30 days.



Figure 44 App Engine Traffic

Figure 44 shows traffic for our API requests.

4. References

- 1- <https://pub.dev/packages/camera>
- 2- <https://api.flutter.dev/flutter/dart-io/ZLibCodec-class.html>
- 3- <https://docs.flutter.dev/cookbook/plugins/picture-using-camera>
- 4- <https://pub.dev/packages/provider>
- 5- <https://pub.dev/packages/http>
- 6- https://pub.dev/packages/flutter_secure_storage
- 7- https://pub.dev/packages/flutter_cache_manager
- 8- <https://eew.com.tr/cloud/cloud.mp4>
- 9- <https://github.com/bartucank/PhotoHocamBackEnd>