```
package com.cybercenter.core.domain.user.service;

import com.cybercenter.core.domain.auth.dto.RegisterRequestDTO;
import com.cybercenter.core.domain.user.entity.Role;
import com.cybercenter.core.infrastructure.dto.MessageTemplate;
import com.cybercenter.core.infrastructure.dto.MessageType;
import com.cybercenter.core.shared.model.StaticMessages;
import com.cybercenter.core.domain.auth.dto.VerifyCodeType;
import com.cybercenter.core.infrastructure.dto.MessageDTO;
import com.cybercenter.core.shared.model.ResponseDTO;
import com.cybercenter.core.domain.user.dto.UserBasicInfoDTO;
import com.cybercenter.core.domain.user.dto.UserInfoDTO;
import com.cybercenter.core.domain.user.dto.UserSearchDTO;
import com.cybercenter.core.domain.user.entity.User;
import com.cybercenter.core.domain.user.entity.UserRole;
import com.cybercenter.core.infrastructure.exception.InvalidVerifyCodeException;
import com.cybercenter.core.infrastructure.exception.NotFoundException;
import com.cybercenter.core.domain.user.mapper.UserMapper;
import com.cybercenter.core.domain.user.repository.UserRepository;
import com.cybercenter.core.domain.user.spe.UserSpecification;
import com.cybercenter.core.shared.utils.VerificationUtils;
import com.cybercenter.core.infrastructure.wrapper.MessageSenderWrapper;
import com.cybercenter.filetoken.FileTokenUtils;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.stereotype.Service;
import org.springframework.util.ObjectUtils;

import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
public class UserService {

    private final UserCacheService userCacheService;
    private final UserMapper userMapper;
    private final UserRepository userRepository;
```

```
    private final VerificationUtils verificationUtils;
    private final RoleCacheService roleCacheService;
    private final FileTokenUtils fileTokenUtils;
    private final MessageSenderWrapper messageSenderWrapper;

    @Value("${fileManager.modelTypeId.user}")
    protected int userModelTypeId;

    public UserInfoDTO findByIdAndScopeId(Long id, Integer scopeId) {
        User user = userCacheService.findById(id);
        if (scopeId != null && user.getRoles().stream().noneMatch(userAuthority ->
Objects.equals(userAuthority.getScopeId(), scopeId))) {
            throw new NotFoundException(StaticMessages.NOT_FOUND_USER);
        }
        return userMapper.toDTO(user);
    }

    /**
     * Retrieves a paginated list of users based on the provided search criteria and pagination
details.
     * This method uses a Specification to filter users in the database based on the search
criteria.
     * It then fetches a paginated list of users from the repository, filters the users based on a
scope ID,
     * converts the list of users to a list of UserInfoDTO objects, and wraps the list of
UserInfoDTO objects
     * in a Page object to maintain pagination information.
     *
     * @param searchDTO the search criteria provided in a UserSearchDTO object.
     * @param pageable  the pagination details, including page number, page size, and sorting
options.
     * @return a Page<UserInfoDTO> object containing the list of UserInfoDTO objects and
pagination information.
     */
    public Page<UserInfoDTO> findAll(UserSearchDTO searchDTO, Pageable pageable) {
        // Create a specification based on the search criteria
        Specification<User> specification = new UserSpecification(searchDTO);

        // Fetch the paginated list of users from the repository
        Page<User> usersPage = userRepository.findAll(specification, pageable);

        // Convert the list of users to a list of UserInfoDTOs
        List<UserInfoDTO> userInfoDTOS = usersPage.getContent().stream()
                .peek(user -> {
```

```
                if (searchDTO.getScopeId() != null) {
                    List<UserRole> userAuthorities = user.getRoles();
                    List<UserRole> list = userAuthorities.stream().filter(userAuthority ->
Objects.equals(userAuthority.getScopeId(), searchDTO.getScopeId())).toList();
                    user.setRoles(list);
                }
            })
            .map(userMapper::toDTO)
            .collect(Collectors.toList());

        // Wrap the list of UserInfoDTOs in a Page object to maintain pagination information
        return new PageImpl<>(userInfoDTOS, pageable, usersPage.getTotalElements());
    }

    public UserInfoDTO save(RegisterRequestDTO dto) {
        return userMapper.toDTO(userCacheService.save(userMapper.ToEntity(dto)));
    }

    /**
     * Updates a user's profile information based on the provided UserInfoDTO.
     * This method handles updating the user's email if it has changed, verifying the user's email if
a verification code is provided,
     * and updating other user information. After updating, it saves the changes to the cache and
returns a ResponseDTO indicating
     * the success of the operation.
     */
    public void updateProfile(long userId, UserInfoDTO userInfoDTO) {
        User user = userCacheService.findById(userId);

        // Update the user's email if it has changed
        updateEmailIfChanged(user, userInfoDTO);

        // Verify the user's email if a verification code is provided
        verifyEmailIfCodeProvided(user, userInfoDTO);

        userMapper.updateUserInfo(user, userInfoDTO);

        userCacheService.save(user);
    }

    /**
     * Updates the user's email if it has changed.
     * Sets the user's email verification status to false if the email is updated.
     *
```

```
     * @param user        The user whose email might be updated.
     * @param userInfoDTO The DTO containing the new email and other user information.
     */
    private void updateEmailIfChanged(User user, UserInfoDTO userInfoDTO) {
        String newEmail = userInfoDTO.getEmail();
        if (!ObjectUtils.isEmpty(newEmail) && !newEmail.equals(user.getEmail())) {
            user.setEmail(newEmail);
            user.setVerifyEmail(false);
        }
    }

    public ResponseDTO sendMailVerifyCode(String mail) {
        Integer verifyCode = verificationUtils.generateVerifyCode(mail,
VerifyCodeType.VERIFY_EMAIL);
        messageSenderWrapper.sendMessage(MessageDTO.builder()
                .to(mail)
                .messageType(MessageType.MAIL.getId())
                .subject("کد تایید ایمیل")
                .template(MessageTemplate.AUTH.VERIFY)
                .tokens(Map.of("verifyCode", String.valueOf(verifyCode)))
                .build());

        return ResponseDTO.success(StaticMessages.SEND_OTP_TO_MAIL);
    }

    /**
     * Verifies the user's email if a verification code is provided.
     * Sets the user's email verification status to true if the verification is successful.
     *
     * @param user        The user whose email might be verified.
     * @param userInfoDTO The DTO containing the verification code and other user information.
     * @throws InvalidVerifyCodeException If the verification code is invalid or does not match the
stored code.
     */
    private void verifyEmailIfCodeProvided(User user, UserInfoDTO userInfoDTO) {
        Integer verifyEmailCode = userInfoDTO.getVerifyEmailCode();
        if (!ObjectUtils.isEmpty(verifyEmailCode)) {
            // Validate the verification code
            verificationUtils.validateVerifyCode(user.getEmail(), verifyEmailCode,
VerifyCodeType.VERIFY_EMAIL);
            user.setVerifyEmail(true);
        }
    }
```

```
    public void updateAuthorities(long userId, int scopeId, List<Integer> newAuthorityIds) {
        User user = userCacheService.findById(userId);

        user.getRoles().removeIf(auth -> auth.getScopeId() == scopeId);

        List<Role> newAuthorities = roleCacheService.findAll().stream()
                .filter(authority -> newAuthorityIds.contains(authority.getId()))
                .toList();

        newAuthorities.forEach(auth -> {
            UserRole ua = UserRole.builder()
                    .user(user)
                    .role(auth)
                    .scopeId(scopeId)
                    .build();
            user.getRoles().add(ua);
        });

        userCacheService.save(user);
    }

    /**
     * Retrieves and returns the profile information of a user as a UserInfoDTO object.
     * This method first attempts to find the user by their unique identifier (userId) in the cache.
     * If the user is not found, a NotFoundException is thrown to indicate that the requested user
profile could not be located.
     *
     * @param userId the unique identifier of the user whose profile information is to be retrieved.
     * @return a UserInfoDTO object containing the user's profile information if the user is found.
     * @throws NotFoundException if the user is not found in the cache.
     */
    public UserInfoDTO getUserProfile(Long userId) {
        User user = userCacheService.findById(userId);
        if (!ObjectUtils.isEmpty(user))
            return userMapper.toDTO(user);
        else
            throw new NotFoundException(StaticMessages.NOT_FOUND_USER);
    }

    public Map<Long, UserBasicInfoDTO> getUserInfo(List<Long> userIds) {
        return userRepository.findBasicInfoByIds(userIds).stream()
                .peek(userBasicInfoDTO -> {
                    String token = userBasicInfoDTO.getCoverImage() != null ?
```

```
                    fileTokenUtils.generateToken(userModelTypeId, null,
userBasicInfoDTO.getId(), userBasicInfoDTO.getCoverImage()) :
                    null;
                userBasicInfoDTO.setFileKey(token);
            })
            .collect(Collectors.toMap(UserBasicInfoDTO::getId, userBasicInfoDTO ->
userBasicInfoDTO));
    }

    public UserBasicInfoDTO getUserInfo(Long userId) {
        UserBasicInfoDTO userBasicInfoDTO = userRepository.findBasicInfoById(userId);
        String token = userBasicInfoDTO.getCoverImage() != null ?
                fileTokenUtils.generateToken(userModelTypeId, null, userBasicInfoDTO.getId(),
userBasicInfoDTO.getCoverImage()) :
                null;
        userBasicInfoDTO.setFileKey(token);
        return userBasicInfoDTO;
    }

    public void addDefaultAuthority(User user, Integer scopeId) {
        Role role = roleCacheService.findByTitle("USER");
        if (userHasAuthority(user, role, scopeId)) {
            return;
        }

        UserRole userRole = UserRole.builder()
                .role(role)
                .user(user)
                .scopeId(scopeId)
                .build();

        user.getRoles().add(userRole);
        userCacheService.save(user);
    }

    public void addDefaultAuthority(Long userId, Integer scopeId) {
        User user = userCacheService.findById(userId);
        addDefaultAuthority(user, scopeId);
    }

    private boolean userHasAuthority(User user, Role role, Integer scopeId) {
        return user.getRoles().stream()
                .anyMatch(userAuthority -> Objects.equals(userAuthority.getRole().getId(),
role.getId()) && Objects.equals(userAuthority.getScopeId(), scopeId));
```

```
    }

    public boolean existUsername(String username) {
        return userRepository.existsByUsername(username);
    }

    public boolean existPhoneNumber(String phoneNumber) {
        return userRepository.existsByPhoneNumber(phoneNumber);
    }
}
```