**2022년도 졸업프로젝트**

주제: TLS를 구현을 통한 홈페이지 제작 및 보안 채널 로그인 기능 구현

|  |  |  |  |
| --- | --- | --- | --- |
| **작성자** | 김도현 | **작성일자** | 2022.05.18 |

* **금일 진행 상황**

|  |  |
| --- | --- |
| **진행 상황** | **비고** |
| **질문 및 진행방식에 대한 논의**  **진행 중**  **완료**   1. **Account 패키지 추가**   **Controller**  @Api(tags = "계좌 생성, 계좌이체, 전계좌 조회") @RestController @RequiredArgsConstructor @RequestMapping("/users/accounts") public class AccountController {  private final AccountService accountService;  private final ResponseService responseService;   @PostMapping(path = "/add")  @ApiOperation(value = "계좌생성")  public BaseResult addAccount(@ApiParam @RequestBody AccountCreateReq accountCreateReq){  try {  accountService.createAccount(accountCreateReq);  return responseService.successResult();  }catch (Exception e){  return responseService.failResult(  e.getMessage()  );  }  }   @PostMapping(path = "/transaction")  @ApiOperation(value="계좌 이체")  public BaseResult transaction(@ApiParam @RequestBody TransactionReq transactionReq){  try {  accountService.transaction(transactionReq);  return responseService.successResult();  }catch (Exception e){  return responseService.failResult(  e.getMessage()  );  }  }   @GetMapping(path = "/inquiry")  @ApiOperation(value = "전계좌 조회")  public BaseResult inquiry(@ApiParam @RequestParam String userId) {  try {  return responseService.listResult(accountService.inquiry(userId));  }catch (Exception e){  return responseService.failResult(  e.getMessage()  );  }  } }  **dto/AccountCreateReq**  @Data @Builder @RequiredArgsConstructor @AllArgsConstructor public class AccountCreateReq {   @ApiModelProperty(required = true)  private String loginId;   @ApiModelProperty(required = true)  private String accountPassword;   public Account toEntity(User User, String encodedPassword, Long balance){  return Account.*builder*()  .userId(User)  .accountPassword(encodedPassword)  .balance(balance)  .CreateDate(new DateConfig().getDateTime())  .build();  } }  **dto/AccountInquiryRes**  @Data @Builder @RequiredArgsConstructor @AllArgsConstructor public class AccountInquiryRes {   private Long accountNumber;  private Long balance; }  **dto/TransactionReq**  @Data @Builder @RequiredArgsConstructor @AllArgsConstructor public class TransactionReq {  @ApiModelProperty(required = true)  private String loginId;   @ApiModelProperty(required = true)  private Long myAccountNumber;   @ApiModelProperty(required = true)  private Long sendAccountNumber;   @ApiModelProperty(required = true)  private String accountPassword;   @ApiModelProperty(required = true)  private Long balance;  }  **Entity**  @Builder @Data @Entity @AllArgsConstructor @NoArgsConstructor public class Account {   @ManyToOne(targetEntity = User.class, fetch = FetchType.*EAGER*)  @JoinColumn(name="userId")  private User userId;   @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  private Long accountNumber;   @Column(columnDefinition = "text",nullable = false)  private String accountPassword;   @Column(nullable = false)  private Long balance;   @Column  private String CreateDate;   public AccountInquiryRes toDto() {  return AccountInquiryRes.*builder*()  .accountNumber(accountNumber)  .balance(balance)  .build();  } }  **Repository**  public interface AccountRepository extends JpaRepository<Account, Long> {  List<Account> findAllByUserId(User userId);   Optional<Account> findAccountByAccountNumber(Long accountNumber);   boolean existsAccountByAccountNumber(Long accountNumber);   @Modifying(clearAutomatically = true)  @Query("update Account a set a.balance = a.balance - ?1 where a.accountNumber = ?2")  void updateMyBalance(Long balance, Long accountNumber);   @Modifying(clearAutomatically = true)  @Query("update Account a set a.balance = a.balance + ?1 where a.accountNumber = ?2")  void updateBalance(Long balance, Long sendAccountNumber); }  **Service**  @Service @RequiredArgsConstructor public class AccountService {  private final AccountRepository accountRepository;  private final UserRepository userRepository;  private final PasswordEncoder passwordEncoder;   @Transactional  public void createAccount(AccountCreateReq accountCreateReq) {  String loginId = accountCreateReq.getLoginId();  String password = accountCreateReq.getAccountPassword();   checkPassword(password);  // 전달 받은 아이디를 통해 사용자가 있는지 확인 없다면 사용자가 없다는 예외를 발생.  if(userRepository.findByLoginId(loginId).isEmpty()){  throw new AccountException(ExceptionMessages.*ERROR\_USER\_NOT\_FOUND*);  }  try{  accountRepository.save(accountCreateReq.toEntity(  userRepository.findByLoginId(loginId).get(),  passwordEncoder.encode(password),  0L));  }catch (Exception e)  {  e.printStackTrace();  throw new AccountException("계좌생성에 실패했습니다.");  }  }   @Transactional  public void transaction(TransactionReq transactionReq)  {  String loginId = transactionReq.getLoginId();  Long sendAccount = transactionReq.getSendAccountNumber();  String password = transactionReq.getAccountPassword();   Account myAccount = accountRepository  .findAccountByAccountNumber(transactionReq.getMyAccountNumber())  .orElseThrow(() -> new AccountException(ExceptionMessages.*ERROR\_ACCOUNT\_NOT\_FOUND*));   // 사용자의 아이디로부터 자신의 계좌가 맞는지 확인.  checkAccount(myAccount, userRepository.findByLoginId(loginId).get());   // 보내는 계좌 번호가 존재하는지 확인.  if(!accountRepository.existsAccountByAccountNumber(sendAccount)){  throw new AccountException(ExceptionMessages.*ERROR\_ACCOUNT\_NOT\_FOUND*);  }  // 사용자의 계좌 비밀번호가 맞는지 확인.  if(!passwordEncoder.matches(password, myAccount.getAccountPassword())) {  throw new AccountException(ExceptionMessages.*ERROR\_ACCOUNT\_PASSWORD\_NOT\_MATCH*);  }  // 사용자의 계좌에 충분한 잔액이 있는지 확인.  if(myAccount.getBalance() < transactionReq.getBalance()) {  throw new AccountException(ExceptionMessages.*ERROR\_ACCOUNT\_BALANCE*);  }  try {  accountRepository.updateMyBalance(transactionReq.getBalance(),  myAccount.getAccountNumber());  accountRepository.updateBalance(transactionReq.getBalance(),  transactionReq.getSendAccountNumber());  }catch (Exception e){  e.printStackTrace();  throw new AccountException("계좌이체에 실패했습니다.");  }  }   @Transactional  public List<AccountInquiryRes> inquiry(String loginId) {  // 정확한 사용자를 넘겨줬는지 확인  if (userRepository.findByLoginId(loginId).isEmpty()) {  throw new AccountException(ExceptionMessages.*ERROR\_USER\_NOT\_FOUND*);  }  return accountRepository  .findAllByUserId(userRepository.findByLoginId(loginId).get())  .stream()  .map(Account::toDto)  .collect(Collectors.*toList*());  }   private void checkPassword(String password)  {  // 계좌 비밀번호는 숫자로 6자로 구성되어있다.  Pattern passwordExpression = Pattern.*compile*("[0-9]{6}");  if(!passwordExpression.matcher(password).matches()){  throw new AccountException(ExceptionMessages.*ERROR\_ACCOUNT\_PASSWORD\_FORMAT*);  }  }   private void checkAccount(Account account, User user){  if(!accountRepository.findAllByUserId(user).contains(account)){  throw new AccountException("전달받은 계좌는 사용자의 계좌가 아닙니다.");  }  } }   1. **JWT 추가**   //@Configuration //@RequiredArgsConstructor // 직접 만든 TokenProvider 와 JwtFilter 를 SecurityConfig 에 적용할 때 사용 //public class JwtSecurityConfig extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> { // private final TokenProvider tokenProvider; // // // TokenProvider 를 주입받아서 JwtFilter 를 통해 Security 로직에 필터를 등록 // @Override // public void configure(HttpSecurity http) { // JwtFilter customFilter = new JwtFilter(tokenProvider); // http.addFilterBefore(customFilter, UsernamePasswordAuthenticationFilter.class); // } //}  **진행 예정** | re |
| * **특이사항 / 협업 사항** | |