**Универзитет Св. Кирил и Методиј - Скопје**

**Факултет за информатички науки и компјутерско инженерство**

**Прв циклус на студии**

**A blue and white logo

Description automatically generated**

**Проектна задача по предмететот**

**Софтверски квалитет и тестирање**

**На тема:**

**Тестирање на апликацијата Time based one-time password**

**Ментор: Кандидат:**

**д-р Ѓорги Маџаров Ѓорги Нечовски**

**м-р Славе Темков број на индекс 211177**

# Вовед

Цел на оваа проектна задача е тестирање на веќе изработена апликација која има главни функционалности како изработка и верифицирање на еднократна лозинка базирана на време. Таа ќе прикаже функционалност преку симулација на мал банкарски систем кој има основни функционалности како автентикација, авторизација на корисници и вработени, транфер, депозит и повлекување на финанскиски средства.

# Организација на апликацијата

Апликацијата е дизајнирана врз шаблонот Model-View-Controller (MVC), што претставува популарен архитектурен модел за развој на софтверски апликации. За backend делот на апликацијата, користи се Spring Framework, кој обезбедува многу функционалности и апстракции за развој на веб апликации, вклучувајќи ги управувањето со зафатниот контролер, сервисите и репозиториумите. Во позадина, апликацијата користи MySQL база на податоци за зачувување на сите релевантни податоци како кориснички информации, трансакции и други податоци поврзани со функционалностите на банкарскиот систем. MySQL е релациона база на податоци која обезбедува стабилност, перформанси и широка поддршка за работа со податоци, што ја прави идеална за користење во банкарските системи и други веб апликации.

За тестирање на апликацијата, се користи H2 база, која е in-memory база главно наменета за тестирање на Java апликации. H2 базата е одлична опција бидејќи обезбедува лесно конфигурирање, брзина, и не бара подесување на физичка база на податоци. Во контекст на тестирање, H2 базата се иницијализира во меморијата и се брише по завршувањето на тестовите, што ја прави идеална за изолирани и автоматизирани тестови.

За единично и интеграциско тестирање на апликацијата, се користи TestNG, која е понова алатка во споредба со JUnit и обезбедува дополнителни функционалности како што се подобра поддршка за групирање тестови, паралелно извршување, и напредни анотации за конфигурација на тестови. Ова го прави TestNG популарен избор за комплексни Java проекти каде е потребна флексибилност при извршување на тестовите и напредни сценарија за тестирање.

За тестирање на MVC делот од апликацијата се користи Spring MVC Test Framework. Овој framework обезбедува можност за тестирање на контролерите, сервисите и други MVC компоненти без потреба од стартување на целосниот веб сервер. Со Spring MVC Test, може да се симулираат HTTP барања и да се провери дали апликацијата се однесува правилно при обработка на тие барања.

Во целост, оваа комбинација на H2 база, TestNG и Spring MVC Test овозможува изолирано, брзо и ефикасно тестирање на апликацијата, покривајќи ја и логиката и презентацискиот слој, со минимално конфигурирање и лесно одржување на тестовите.

Тестирањето е поделено на 3 главни делови:

* Тестирање на repositories
* Тестирање на сервиси
* Тестирање на контролери

# Тестирање

## Тестирање на repositories

Тестирањето на repositories е всушност најлесниот дел од проектот. Тие се веќе вграден и тестиран дел од Spring, па потребно е само да ги имплементираме и да се осигураме дека нашите модели се креираат според нашите барања. Тестирањето на овој проект се одвива само со TestNG. Тука нема да ги опфатам сите тестови, бидејќи сите се одваат на исти начин, **односно креирај модел -> зачувај во база -> провери дали е успешно зачуван**

Ќе прегледаме неколку такви тестови, прв е креирање на корисник на апликацијата:

*@Test*(groups = "repository tests")  
void testCreateCustomerRepository() {  
 Customer customer = new Customer("Gjore", "Neco", "111", "111", "111".getBytes(), "populargjorgi@gmail.com", "111", "111".getBytes());  
 customerRepository.save(customer);  
  
 Customer savedCustomer = customerRepository.findByTransactionNumber("111");  
 Assert.**assertNotNull**(savedCustomer, "Saved customer should not be null");  
}

Како што спомнавме погоре, се креира корисник, се зачувува и се прегледува дали постои, ако не постои се фрла погрешен тип на резултат односно нашиот тест паѓа.

A diagram of a customer

Description automatically generated

На овој начин се тестираат сите repository функции и се прилично лесни да се тестираат поради тоа. Затоа нема да се задржам многу долго на нив и ќе преминиме на поглавните делови од апликацијата, односно сервисите.

## Тестирање на сервиси

Сервисите се малку покомплицирани за тестирање во споредба со repositories бидејќи тие не се вградени во Spring туку тие ги креираме ние. Затоа треба да се обрне повеќе внимание при извршување на истите. Во овој проект се осигурав дека што повеќе се имплементирани SOLID принципи за да може сервисите да бидат што можно полесно тестирани во своите методи (освен оние методи кои се невозможни, поради потреба на интеракција со корисник).

За почеток ќе започниме со сервисот што го генерира One Time Password врз кој работи целата апликација, поради што генерирањето на сервисот е главно опфатено математика, тој има само еден метод generateTOTP(). За него имаме 2 тестови, еден за проверка кога генерира точна лозинка, а еднаш кога генерира грешна:

*@Test*(groups = "one time password service")  
void testVerifyTOTPTrue() {  
 Customer customer1 = new Customer("Gjore", "Neco", "111", "111", "111".getBytes(), "populargjorgi@gmail.com", "111", "111".getBytes());  
 customerRepository.save(customer1);  
  
 Customer savedCustomer = customerRepository.findByTransactionNumber(customer1.getTransactionNumber());  
  
 String otp = oneTimePasswordService.generateTOTP(savedCustomer.getSecretKey(), LocalDateTime.**now**());  
 Assert.**assertTrue**(oneTimePasswordService.verifyTOTP(savedCustomer.getSecretKey(), otp, LocalDateTime.**now**()), "OTP should be valid");  
}  
  
*@Test*(groups = "one time password service")  
void testVerifyTOTPFalse() {  
 Customer customer1 = new Customer("Gjore", "Neco", "111", "111", "111".getBytes(), "populargjorgi@gmail.com", "111", "111".getBytes());  
 customerRepository.save(customer1);  
  
 Customer savedCustomer = customerRepository.findByTransactionNumber(customer1.getTransactionNumber());  
  
 String otp = oneTimePasswordService.generateTOTP(savedCustomer.getSecretKey(), LocalDateTime.**now**());  
 Assert.**assertFalse**(oneTimePasswordService.verifyTOTP(savedCustomer.getSecretKey() +"1", otp, LocalDateTime.**now**()), "OTP should be valid");  
}

Нареден сервис за тестирање е сервисот за трансакција, овој е малку покомплициран, поради што тој ги контролира сите трансакции кои се одвиваат во банката. Како прв метод што ќе го тестираме е креирање на трансакција помеѓу 2 корисници, тој го има следниот изглед:

/\*\*  
 \* Transfers a specified amount from one customer to another.  
 \*  
 \* @param *fromUser* The customer initiating the transfer.  
 \* @param *to* The recipient of the transfer.  
 \* @param *amount* The amount to transfer.  
 \* @return The transaction object representing the transfer.  
 \* @throws InsufficientMoneyException Thrown if the initiating customer has insufficient funds.  
 \* @throws InvalidTransferNumberException Thrown if the transfer number is invalid.  
 \* @throws NegativeAmountException Thrown if the transfer amount is negative.  
 \* @throws SameUserException Thrown if the recipient is the same as the sender.  
 \*/  
Transaction transfer(Customer *fromUser*, String *to*, double *amount*) throws InsufficientMoneyException, InvalidTransferNumberException, NegativeAmountException, SameUserException;

*@Override  
@Transactional*public Transaction transfer(Customer *fromUser*, String *to*, double *amount*) throws InsufficientMoneyException, InvalidTransferNumberException, NegativeAmountException, SameUserException {  
 Customer toUser = customerRepository.findByTransactionNumber(*to*);  
 if (toUser == null){  
 throw new InvalidTransferNumberException();  
 }  
 if (*fromUser*.getBalance() < *amount*){  
 throw new InsufficientMoneyException();  
 }  
 if (*amount* <= 0){  
 throw new NegativeAmountException();  
 }  
 if (*fromUser*.getTransactionNumber().equals(*to*)){  
 throw new SameUserException();  
 }  
  
 LocalDateTime transactionTime = LocalDateTime.**now**();  
  
 String oneTimePassword = oneTimePasswordService.generateTOTP(*fromUser*.getSecretKey(), transactionTime);  
  
 Transaction transaction = new Transaction(*fromUser*, toUser, *amount*, transactionTime, TransactionType.TRANSFER);  
 emailService.generatePasswordEmail(*fromUser*.getEmail(), oneTimePassword);  
  
 transactionRepository.save(transaction);  
  
 return transaction;  
}

A diagram of a diagram

Description automatically generated

Од оваа функција произлегуваат 5 различни тестови, односно 4 за секој вид на Exception и еден за кога функцијата успешно е извршена без проблеми. Тука ќе поставам еден таков тест, додека останатите ќе бидат во кодот.

*@Test*(groups = "transfer service")  
public void testCreateTransactionInsufficientMoney() {  
 Customer customer1 = new Customer("Gjore", "Neco", "111", "111", "111".getBytes(), "populargjorgi@gmail.com", "111", "111".getBytes());  
 customer1.setBalance(100);  
 Customer customer2 = new Customer("Gjore1", "Neco1", "222", "222", "222".getBytes(), "gjorginechovski@gmail.com", "222", "222".getBytes());  
  
 customerRepository.save(customer1);  
 customerRepository.save(customer2);  
  
 try {  
 transferService.transfer(customer1, customer2.getTransactionNumber(), 150);  
 } catch (InsufficientMoneyException *e*) {  
 Assert.**assertTrue**(true);  
 return;  
 }  
 catch (Exception *e*){  
 Assert.**fail**("Exception should not have been caught here", *e*);  
 }  
  
 Assert.**fail**("InsufficientMoneyException should have been thrown");  
}

Што може да се забележи? Тестот започнува со креирање на 2 корисници, на кој на првиот му се задава баланс од 100. Но кога се обидeме да направиме трансфер на 150 очекуваме да се фрли Exception. Тестот е единствено успешен ако успешно се фрли и фати тој Exception. Ако методот за тестирање успее да стигни до крајот на методот има Assert.fail() кој што го паѓа тестот бидејќи трансакцијата пoминала без проблем. Сите тестови за овој метод изгледаат слично.

Нареден метод за тестирање е методот со кој ја завршуваме трансакцијата, односно се префрлуваат парите од еден корисник на другиот, и трансакцијата преминува во ЗАВРШЕНА состојба.

/\*\*  
 \* Completes a transfer transaction.  
 \*  
 \* @param *transaction* The transaction to be completed.  
 \* @throws TimeExceededException Thrown if the transaction completion time has exceeded.  
 \*/  
void finishTransfer(Transaction *transaction*) throws TimeExceededException;

Овој метод е сличен како минатиот, само што тој има помал број на можни фрлени Exceptions.

Наредно за тестирање е автентикацискиот сервис. Овој сервис има 2 главни методи, register() и login(). Исто како сервисто за трансакција овој сервис е многу важен бидејќи една грешка, и може некој да влези во име на друг корисник. Затоа сервисот мора да биде тестиран во длабочина. Ќе започнеме со register()

/\*\*  
 \* Registers a new customer with the provided information.  
 \*  
 \* @param *name* The customer's first name.  
 \* @param *surname* The customer's last name.  
 \* @param *transactionNumber* The unique transaction number to be associated with the customer.  
 \* @param *email* The customer's email address.  
 \* @param *password* The desired password for the new account.  
 \* @param *repeatPassword* The repeated password for confirmation.  
 \* @throws EmailExistsException Thrown if the provided email address already exists in the system.  
 \* @throws PasswordsDontMatchException Thrown if the provided passwords do not match.  
 \* @throws TransactionNumberExistsExceptions Thrown if the provided transaction number already exists in the system.  
 \*/  
void register(String *name*, String *surname*, String *transactionNumber*, String *email*, String *password*, String *repeatPassword*) throws EmailExistsException, PasswordsDontMatchException, TransactionNumberExistsExceptions, PasswordNotMatchingRequirementsException;

Овој метод би се тестирал многу слично како методот за трансакција и неговиот граф би изледал:

A diagram of a diagram

Description automatically generated

Другите методи на сервисот за автентикација се многу слични како останатите тестирани методи во сите сервиси (сите имаат слична структура).

## Тестирање контролери

За тестирање на контролерите потребно е да се воведи нова алатка тоа е Spring MVC test која всушност е framework вграден во Spring framework и овозможува тестирање на MVC endpoints без потреба на корисник, односно самиот developer може да тестира само со код. Овозможува креирање на Mock повици, Mock сесии и да се предаваат параметри кои моделот треба да ги користи во View-то.

За почеток да видиме еден лесен пример за таков тест.

*@Test*(groups = "authentication controller")  
void testLoginSuccess() throws Exception {  
 String name = "Gjore";  
 String surname = "Neco";  
 String transactionNumber = "111";  
 String password = "Test123!";  
 String repeatPassword = "Test123!";  
 String email = "populargjorgi@gmail.com";  
  
 authenticationService.register(name, surname, transactionNumber, email, password, repeatPassword);  
  
 Customer savedCustomer = customerRepository.findByTransactionNumber(transactionNumber);  
 savedCustomer.setActivatedAccount(true);  
 customerRepository.save(savedCustomer);  
  
 mockMvc.perform(**post**("/login")  
 .param("transactionNumber", transactionNumber)  
 .param("password", password)  
 .session(new MockHttpSession())  
 .contentType(MediaType.APPLICATION\_FORM\_URLENCODED))  
 .andExpect(**status**().is3xxRedirection())  
 .andExpect(**view**().name("redirect:/home"))  
 .andExpect(**cookie**().exists("token"))  
 .andExpect(**model**().attributeDoesNotExist("error"))  
 .andDo(**print**());  
}

За тестот се креира корисник како на тестовите досега, и потоа настапува тестирањето на MVC. Односно се кажува каков request планираме да пуштиме на кој endpoint, и кои информации сакаме да ги пуштиме. Поради што апликацијата работи со сесија, потребно е и да и се задади MockHttpSession, во која ќе складира најавен корисник и други потребни информации. Потоа со andExpect() се кажува што се очекува да ни врати контролерот во овој случај кажуваме дека сакаме 3xxRedirection поради што корисникот упешно се најавува и се префрла на /home. Потоа се очекува сесијата да содржи token со која знае дека корисникот е автентициран.

Методот за креирање на трансфер помеѓу 2 корисници веќе беше спомнат, но сега како би изгледал истиот кога би се имплементирал со помош на MVC тестирање. Прво да видиме како изгледа endpoint-от во Spring:

*@PostMapping*("/transfer")  
public String transfer(*@RequestParam* String *transferNumber*,  
 *@RequestParam* double *amount*,  
 Model *model*,  
 HttpSession *session*){  
 Customer user = (Customer) *session*.getAttribute("user");  
 Transaction transaction = null;  
 try {  
 transaction = this.transferService.transfer(user,*transferNumber*,*amount*);  
 } catch (InsufficientMoneyException | InvalidTransferNumberException | NegativeAmountException |  
 SameUserException *e*) {  
 *model*.addAttribute("error", *e*.getMessage());  
 *model*.addAttribute("template", "transactions/transfer");  
  
 return "master";  
 }  
  
 *model*.addAttribute("transferNumber", *transferNumber*);  
 *model*.addAttribute("amount", *amount*);  
 *model*.addAttribute("template","transactions/transferPasswordInput");  
 *model*.addAttribute("transactionId", transaction.getId());  
 return "master";  
}

Може да приметиме дека функцијата очекува 2 задолжителни параметри, transferNumber и amount. А потоа користи модел и сесија за работа на своите функции. Тестирањето на овој endpoint би изгледал вака:

*@Test*(groups = "transfer controller")  
public void testTransferSuccess() throws Exception {  
 MockHttpSession session = setupCustomerSession();  
 Customer customer2 = new Customer("Gjore1", "Neco1", "222", "222", "222".getBytes(), "gjorginechovski@gmail.com", "222", "222".getBytes());  
 customerRepository.save(customer2);  
  
 String token = (String) session.getAttribute("token");  
 Cookie tokenCookie = new Cookie("token", token);  
  
 String transferNumber = "222";  
 double amount = 50.0;  
  
 mockMvc.perform(**post**("/transfer")  
 .session(session)  
 .cookie(tokenCookie)  
 .param("transferNumber", transferNumber)  
 .param("amount", String.**valueOf**(amount)))  
 .andExpect(**status**().isOk())  
 .andExpect(**view**().name("master"))  
 .andExpect(**model**().attributeExists("template"))  
 .andExpect(**model**().attribute("template", "transactions/transferPasswordInput"))  
 .andExpect(**model**().attribute("transferNumber", transferNumber))  
 .andExpect(**model**().attribute("amount", amount))  
 .andExpect(**model**().attributeExists("transactionId"))  
 .andDo(**print**());  
}

Се креираат 2 корисници, и се превзема креираното cookie за одржување на сесија. Потоа за тестирање повторно се пушта POST request на /transfer каде се предава токенот за сесија исто и cookie-to што корисникот треба да го има со секој request. Наредно се испраќаат 2та задолжителни параметри и се пишуваат резултатите кои се очекуваат од овој request. Статусот да биде 200, да имаме template со име master, и во него да содржи друг template transferPasswordInput. А потоа се очекуваат останатите параметри потребни за да може да се прикажи View-то што го враќа функцијата.

На овој начин прилично сите Controller методи се тестираат, само со некои измени на пчекуваните параметри и Views.

# Заклучок

За една апликација да биде лесно тестирана и да има покриеност на сите нејзини компоненти, неопходно е да се следат SOLID принципите. Овие принципи обезбедуваат дека кодот е модуларен, лесен за разбирање и одржување, што директно влијае на квалитетот и ефикасноста на тестирањето.

Тестирањето на апликацијата носи значајни придобивки. Прво, тоа го намалува ризикот од воведување грешки во делови кои инаку би можеле да останат незабележани. Во комплексни системи, дури и малите промени можат да предизвикаат неочекувани проблеми, но со добра покриеност на тестовите, овие ризици се сведуваат на минимум. Ова е особено важно при развој и надградба на апликацијата, кога секоја измена може да има несакани последици врз другите делови од системот.

Дополнително, тестирањето обезбедува сигурност при одржување и промена на кодот. Unit тестовите, на пример, осигуруваат дека промените во еден дел од кодот нема да ја нарушат функционалноста на други делови. Ова е клучно за долгорочно одржување на апликацијата, бидејќи овозможува брзо и лесно идентификување на проблемите и спречување на потенцијални дефекти пред тие да стигнат до продукциска средина.

Spring framework овозможува имплементација на многу алатки за тестирање на апликации, кои го олеснуваат процесот на тестирање и го прават поефикасен. Две од најпопуларните алатки кои Spring ги нуди за тестирање се опфатени во оваа семинарска работа. Преку нив, може да се изврши тестирање на различни аспекти на апликацијата, од тестирање на бизнис логика до интеграционо тестирање на цели модули, што овозможува длабока и детална анализа на квалитетот на кодот.

Со правилно применување на овие алатки и принципи, апликацијата станува не само потрајна и поотпорна на грешки, туку и многу полесна за понатамошен развој и одржување. Тестирањето станува интегрален дел од процесот на развој, обезбедувајќи високо ниво на доверба во стабилноста и квалитетот на финалниот продукт.

Линк до github repo:

https://github.com/GjorgiNechovski/Time-Based-One-Time-Password