UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Proiect SCTI

Student: Péntek Tamás
Grupa: 30642
Data: 14.01.2021

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Introducere

Testarea software este o etapa foarte importanta in dezvoltarea aplicatiilor, pentru ca ofera partilor interesate informatii referitoare la calitatea produsului. Pentru aceasta tema, am ales ca si punct de pornire proiectul facut la materia Sisteme Distribuite din acest semestru.

Obiectivele proiectului

Acest proiect consta intr-o platforma online de medicamente, conceputa pentru a gestiona pacientii, ingrijitorii si medicamentele. Sistemul poate fi accesat de trei tipuri de utilizatori dupa autentificare: medic, pacient si ingrijitor.

Medicul poate efectua operatii CRUD pe conturile pacientului, pe conturile ingrijitorului si pe lista medicamentelor disponibile in sistem. Fisa medicala a unui pacient trebuie sa contina o descriere a starii medicale a pacientului. Pe langa acesta, medicul poate crea un plan de medicamente pentru un pacient, constand dintr-o lista de medicamente si intervale de administrare care trebuie luate zilnic, si perioada tratamentului. Pacientii isi pot vizualiza conturile si planurile de medicamente. Ingrijitorii isi pot vizualiza pacientii asociati si planurile de medicamente corespunzatoare.

Aceasta aplicatie este una mai complexa, are trei parti principale: baza de date, backend si frontend. Comunicarea dintre frontend si backend se face prin servicii REST, utilizand metode HTTP precum POST si GET. Aplicatia este deployed pe cloud-ul Heroku, dar poate fi rulata si testata si de pe localhost.

Baza de date folosita este Postgres, care este un server de baze de date open source si are un add-on pentru Heroku Cloud, deci este util si atunci cand aplicatia va fi deployed pe cloud.

Backend-ul, adica serverul web a aplicatiei este implementat folosind Spring Boot. Acest framework genereaza toate configuratiile necesare, utilizate de server si in acest fel dezvoltatorul poate sa concentreze pe lucruri mai importante.

Frontend-ul este implementat in React, care genereaza fisiere JavaScript si HTML care sunt utilizate pentru a afisa date intr-un browser web. React este o biblioteca JavaScript open-source, pentru construirea de interfete utilizator sau componente UI si este intretinuta de Facebook. In frontend-ul aplicatiei sunt create o multime de componente React pentru a avea o vizualizare mai buna a datelor din baza de date relationala.

Testare unitara

Testarea la nivel de unitate (Unit testing) este o strategie foarte eficienta a dezvoltatorilor profesionisti, deoarece asigura ca fiecare sarcina este realizata corect, codul scris functioneaza conform presupunerilor. Pentru un proiect complex, este foarte important folosirea testarii la nivel de unitate, pentru ca previne aparitia unor erori, furnizeaza un castig de timp in favoarea altor activitati de testare si o documetatie a codului si usureaza efectuarea schimbarilor in proiect.

Un alt avantaj a testelor la nivel de unitate este ca sunt rulate in izolare, adica testeaza o singura clasa sau o singura metoda, si pe langa aceasta sunt foarte rapide, un singur unit test dureaza doar cateva milisecunde.

In zilele de azi, unit tests sunt scrise folosind niste framework-uri de test care usureaza scrierea de teste pentru programatori. In cadrul acestui proiect am folosit doua framework-uri pentru a realiza testele: JUnit si Mockito.

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Mockito este un framework de testare open-source pentru Java lansat sub licenta MIT. Acest framework permite crearea de obiecte mock in teste unitare automatizate. Scopul obiectelor mock este de a verifica iesirea indirecta, aceste obiecte muta focusul unui unit test de la stare la comportament. Un obiect mock este o implementare falsa (dummy) pentru o interfata sau o clasa in care sunt definite iesirea anumitor apeluri de metoda. Obiectele mock sunt configurate pentru a efectua un anumit comportament in timpul unui test.

JUnit este tot un framework de testare unitara pentru limbajul de programare Java. Acesta joaca un rol crucial in dezvoltarea testata. JUnit ofera adnotari pentru identificarea metodelor de testare, assert-uri pentru testarea rezultatelor asteptate, runners pentru rularea testelor. Assert-urile furnizeaza o modalitate standard pentru a exprima rezultatul testului, daca testul s-a terminat cu succes sau nu.

In cadrul acestui proiect, framework-ul Mockito a fost folosit pentru a crea niste obiecte mock cu o implementare dummy, iar framework-ul JUnit a fost folosit pentru assert-uri si pentru a rula metodele de test.

In unit testele realizate in acest proiect, sunt verificate urmatoarele:

- adaugarea, actualizarea si stergerea unui cont in/din baza de date
- gasirea tuturor conturilor din baza de date
- gasirea unui cont dupa ID sau dupa nume de utilizator si parola
- adaugarea unui utilizator de tip doctor in baza de date
- gasirea unui utilizator de tip doctor dupa ID-ul de cont
- gasirea unui plan de medicamente pe baza ID-ului

Din proiectul prezentat mai sus, au fost testate 3 clase. Cele trei clase sunt de tip Service, fiecare comunica cu baza de date folosind niste clase de tip Repository. In fiecare clasa de test sunt definite doua atribute: un obiect de Repository si un obiect de Service.

In cazul obiectului de Repository, este folosit adnotarea @Mock, care este cea mai utilizata adnotare din framework-ul Mockito. @Mock este folosit pentru a crea si a injecta o instanta de tip mock fara a fi nevoie sa apelam manual metoda Mockito.mock.

Adnotarea @InjectMocks este utilizata pentru a injecta automat instanta de tip mock in obiectul testat. Prima clasa de test are ca si denumire AccountServiceTest, in aceasta clasa am testat metodele folosite in clasa AccountService. Aceasta clasa contine metode de tip CRUD referitoare la clasa de Account.

Clasa AccountServiceTest:

```
@RunWith(MockitoJUnitRunner.class)
public class AccountServiceTest {

    @Mock
    AccountRepository accountRepository;

    @InjectMocks
    AccountService accountService;

    @Test
    public void findAccountsTest() {
        Account account1 = new Account(UUID.randomUUID(), "user1", "password1", AccountType.PATIENT);
        Account account2 = new Account(UUID.randomUUID(), "user2", "password2", AccountType.CAREGIVER);
        Account account3 = new Account(UUID.randomUUID(), "user3", "password3", AccountType.DOCTOR);
        List<Account> accountList = Arrays.asList(account1, account2, account3);
        List<AccountDTO> accountListFinal = Arrays.asList(AccountBuilder.toAccountDTO(account1),
AccountBuilder.toAccountDTO(account2), AccountBuilder.toAccountDTO(account3));
```

```
        when(accountRepository.findAll()).thenReturn(accountList);
        Assert.assertEquals(accountListFinal, accountService.findAccounts());
        Assert.assertEquals(accountListFinal.get(0), accountService.findAccounts().get(0));
    }

    @Test
    public void findAccountByIdTest() {
        UUID accountID = UUID.randomUUID();
        UUID wrongAccountID = UUID.randomUUID();
        Account account1 = new Account(accountID, "user1", "password1", AccountType.PATIENT);
        when(accountRepository.findById(accountID)).thenReturn(Optional.of(account1));
        Assert.assertEquals(AccountBuilder.toAccountDTO(account1), accountService.findAccountById(accountID));
        Assert.assertThrows(ResourceNotFoundException.class, () -> accountService.findAccountById(wrongAccountID));
    }

    @Test
    public void findAccountByUsernameAndPasswordTest() {
        UUID accountID = UUID.randomUUID();
        Account account1 = new Account(accountID, "user1", "password1", AccountType.PATIENT);
        when(accountRepository.findAccountByUserNameAndPassword(account1.getUserName(),
account1.getPassword())).thenReturn(Optional.of(account1));
        Assert.assertEquals(AccountBuilder.toAccountDTO(account1),
accountService.findAccountByUsernameAndPassword(account1.getUserName(), account1.getPassword()));
        Assert.assertThrows(ResourceNotFoundException.class, () ->
accountService.findAccountByUsernameAndPassword("wrongUsername", "wrongPassword"));
    }

    @Test
    public void insertAccountTest() {
        UUID accountID = UUID.randomUUID();
        Account account = new Account(accountID, "user1", "password1", AccountType.CAREGIVER);
        AccountDTO accountDTO = AccountBuilder.toAccountDTO(account);
        when(accountRepository.save(Mockito.any(Account.class))).thenReturn(account);
        Assert.assertEquals(account.getId(), accountService.insertAccount(accountDTO));
    }

    @Test
    public void updateAccountTest() {
        UUID accountID = UUID.randomUUID();
        UUID wrongAccountID = UUID.randomUUID();
        Account account = new Account(accountID, "user1", "password1", AccountType.PATIENT);
        AccountDTO accountDTO = AccountBuilder.toAccountDTO(account);
        when(accountRepository.findById(accountID)).thenReturn(Optional.of(account));
        when(accountRepository.save(Mockito.any(Account.class))).thenReturn(account);
        Assert.assertEquals(account.getId(), accountService.updateAccount(accountID, accountDTO));
        Assert.assertThrows(ResourceNotFoundException.class, () -> accountService.updateAccount(wrongAccountID,
accountDTO));
    }

    @Test
    public void deleteAccountTest() {
        UUID accountID = UUID.randomUUID();
        doNothing().when(accountRepository).deleteById(accountID);
        accountService.deleteAccount(accountID);
        verify(accountRepository, times(1)).deleteById(accountID);
    }
}
```

Codul din clasa AccountService pe care au fost aplicate testele:

```java
@Service
public class AccountService {

    private final AccountRepository accountRepository;

    @Autowired
    public AccountService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    public List<AccountDTO> findAccounts() {
        List<Account> accountList = accountRepository.findAll();
        return accountList.stream()
                .map(AccountBuilder::toAccountDTO)
                .collect(Collectors.toList());
    }

    public AccountDTO findAccountById(UUID accountId) {
        Optional<Account> foundAccount = accountRepository.findById(accountId);
        if (!foundAccount.isPresent()) {
            throw new ResourceNotFoundException(Account.class.getSimpleName() + " with id: " + accountId);
        }
        return AccountBuilder.toAccountDTO(foundAccount.get());
    }


    public AccountDTO findAccountByUsernameAndPassword(String username, String password) {
        Optional<Account> foundAccount = accountRepository.findAccountByUserNameAndPassword(username,
password);
        if (!foundAccount.isPresent()) {
            throw new ResourceNotFoundException(Account.class.getSimpleName() + " with id: " + username);
        }
        return AccountBuilder.toAccountDTO(foundAccount.get());
    }

    public UUID insertAccount(AccountDTO accountDTO) {
        Account account = AccountBuilder.toAccountEntity(accountDTO);
        account = accountRepository.save(account);
        return account.getId();
    }

    public UUID updateAccount(UUID accountId, AccountDTO accountDTO) {
        Optional<Account> foundAccount = accountRepository.findById(accountId);
        if (!foundAccount.isPresent()) {
            throw new ResourceNotFoundException(Account.class.getSimpleName() + " with id: " + accountId);
        }
        Account account = foundAccount.get();
        account.setId(accountDTO.getId());
        account.setAccountType(accountDTO.getAccountType());
        account.setUserName(accountDTO.getUserName());
        account.setPassword(accountDTO.getPassword());
        account = accountRepository.save(account);
        return account.getId();
    }

    public void deleteAccount(UUID accountId) {
        accountRepository.deleteById(accountId);
    }
}
```

A doua clasa de test este DoctorServiceTest, aici am testat metodele din clasa DoctorService: o metoda care adauga in baza de date un utilizator de tip doctor si inca o metoda care cauta un utilizator de tip doctor dupa ID-ul de cont.

Clasa DoctorServiceTest:

```
@RunWith(MockitoJUnitRunner.class)
public class DoctorServiceTest {

    @Mock
    AccountRepository accountRepository;
    @Mock
    DoctorRepository doctorRepository;


    @InjectMocks
    DoctorService doctorService;

    @Test
    public void insertDoctorTest() {
        UUID doctorID = UUID.randomUUID();
        Account account = new Account(UUID.randomUUID(), "doctor", "password", AccountType.DOCTOR);
        Doctor doctor = new Doctor(doctorID, account);

        when(accountRepository.save(Mockito.any(Account.class))).thenReturn(account);
        when(doctorRepository.save(Mockito.any(Doctor.class))).thenReturn(doctor);
        Assert.assertEquals(doctor.getId(), doctorService.insertDoctor(DoctorBuilder.toDoctorDTO(doctor)));
    }

    @Test
    public void findDoctorByAccountIdTest() {
        UUID doctorID = UUID.randomUUID();
        Account account = new Account(UUID.randomUUID(), "doctor", "password", AccountType.DOCTOR);
        Doctor doctor = new Doctor(doctorID, account);
        DoctorDTO doctorDTO = DoctorBuilder.toDoctorDTO(doctor);
        when(doctorRepository.findDoctorByAccount_Id(account.getId())).thenReturn(Optional.of(doctor));
        Assert.assertEquals(doctorDTO, doctorService.findDoctorByAccountId(account.getId()));
        Assert.assertThrows(ResourceNotFoundException.class, () ->
doctorService.findDoctorByAccountId(UUID.randomUUID()));
    }
}
```

Codul din clasa DoctorService pe care au fost aplicate testele:

```
@Service
public class DoctorService {

    private final DoctorRepository doctorRepository;
    private final AccountRepository accountRepository;

    @Autowired
    public DoctorService(DoctorRepository doctorRepository, AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
        this.doctorRepository = doctorRepository;
    }
```

```
    public UUID insertDoctor(DoctorDTO doctorDTO) {
        Doctor doctor = DoctorBuilder.toDoctorEntity(doctorDTO);
        Account account = doctor.getAccount();
        accountRepository.save(account);
        doctor = doctorRepository.save(doctor);
        return doctor.getId();
    }

    public DoctorDTO findDoctorByAccountId(UUID accountId) {
        Optional<Doctor> foundDoctor = doctorRepository.findDoctorByAccount_Id(accountId);
        if (!foundDoctor.isPresent()) {
            throw new ResourceNotFoundException(Doctor.class.getSimpleName() + " with id: " + accountId);
        }
        return DoctorBuilder.toDoctorDTO(foundDoctor.get());
    }
}
```

Ultima clasa de test este MedicationPlanServiceTest in care este testata o metoda din clasa MedicationPlanService: metoda cauta un plan de medicamente dupa ID.

Clasa MedicationPlanServiceTest:

```
@RunWith(MockitoJUnitRunner.class)
public class MedicationPlanServiceTest {

    @Mock
    MedicationPlanRepository medicationPlanRepository;

    @InjectMocks
    MedicationPlanService medicationPlanService;

    @Test
    public void findMedicationPlanByIdTest() {
        UUID medicationPlanID = UUID.randomUUID();
        UUID wrongMedicationID = UUID.randomUUID();
        Medication medication1 = new Medication(UUID.randomUUID(), "Algocalmin", "No side effects", "2/day");
        Medication medication2 = new Medication(UUID.randomUUID(), "Paracetamol", "No side effects", "3/day");
        MedicationPlan medicationPlan = new MedicationPlan(medicationPlanID, 24, Arrays.asList(medication1,
medication2));
        when(medicationPlanRepository.findById(medicationPlanID)).thenReturn(Optional.of(medicationPlan));
        MedicationPlanDTO medicationPlanResult = medicationPlanService.findMedicationPlanById(medicationPlanID);
        Assert.assertEquals(MedicationPlanBuilder.toMedicationPlanDTO(medicationPlan).getId(),
medicationPlanResult.getId());
        Assert.assertEquals(MedicationPlanBuilder.toMedicationPlanDTO(medicationPlan).getTreatmentPeriod(),
medicationPlanResult.getTreatmentPeriod());
        Assert.assertEquals(MedicationPlanBuilder.toMedicationPlanDTO(medicationPlan).getMedications(),
medicationPlanResult.getMedications());
        Assert.assertThrows(ResourceNotFoundException.class, () ->
medicationPlanService.findMedicationPlanById(wrongMedicationID));
    }
}
```

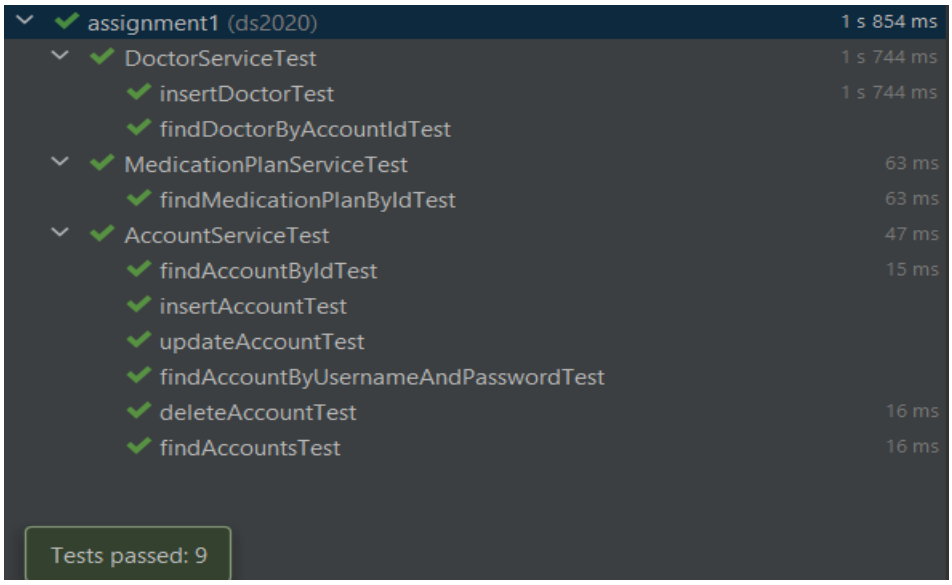
Codul din clasa MedicationPlanService pe care au fost aplicate testele:

```java
@Service
public class MedicationPlanService {

    private final MedicationPlanRepository medicationPlanRepository;

    @Autowired
    public MedicationPlanService(MedicationPlanRepository medicationPlanRepository) {
        this.medicationPlanRepository = medicationPlanRepository;
    }

    public MedicationPlanDTO findMedicationPlanById(UUID medicationPlanId) {
        Optional<MedicationPlan> foundMedicationPlan = medicationPlanRepository.findById(medicationPlanId);
        if (!foundMedicationPlan.isPresent()) {
            throw new ResourceNotFoundException(MedicationPlan.class.getSimpleName() + " with id: " +
medicationPlanId);
        }
        return MedicationPlanBuilder.toMedicationPlanDTO(foundMedicationPlan.get());
    }
}
```

Pentru verificarea si evaluarea testelor am folosit clasa Assert din framework-ul JUnit. Metoda *Assert.assertEquals(valoare asteptata, valoare primita)* verifica daca cei doi parametri sunt egale sau nu. Aceasta metoda a fost folosita in toate metodele de test. O alta metoda, tot din framework-ul Assert, a fost folosit in anumite cazuri (unde am avut in codul sursa o aruncare de exceptie): *Assert.assertThrows(tipul exceptiei, metoda care arunca o exceptie)* verifica daca al doilea parametru (o metoda) arunca o exceptie de tipul primului parametru. In acest fel a fost testat fiecare ramura dintr-o metoda: si cand are un rezultat valid, si cand arunca o exceptie.

Un alt tip de verificare am folosit in clasa AccountServiceTest la metoda deleteAccountTest. Aceasta metoda testeaza stergerea unui cont si fiindca metoda deleteAccount este de tip void, nu pot sa testez cu Assert. Din aceasta cauza am folosit metoda *verify(obiect mock, numarul de apeluri).nume_de_metoda* din framework-ul Mockito, care verifica daca metoda a fost apelata de o data sau nu.

Rezultatele testarilor la nivel de unitate sunt prezentate mai jos:



In cazul fiecarui test au fost generate niste date de test, iar aceste data au avut rolul de a inlocui datele din baza de date, ca sa nu trebuie sa accesam baza de date, sa nu avem dependite cu baza de date, doar sa testam fiecare metoda separat.

Aceste teste au fost foarte ajutatoare in timpul scrierii codului, deoarece in acest fel am reusit sa detectez si sa elimin mult mai repede defectele in ciclul de dezvoltare a acestui proiect si in acest fel scrierea codului a fost mult mai eficient. Unit testele vor fi folositoare si in viitor, daca vor aparea niste schimbari, optimizari in codul sursa a metodelor. Un alt avantaj este ca unit testing furnizeaza si o documentare a codului sursa, daca acest proiect va ajunge la o alta echipa de programatori, vor intelege mult mai usor codul sursa.

Un lucru pe care am observat ca afecteaza testabilitatea si durata testarii este ca daca folosesc mai multe obiecte de tip mock intr-o metoda, atunci timpul de executie a metodei de test creste destul de mult. Totusi, consider ca acest lucru este normal, fiindca in spate framework-ul de Mockito initializeaza mai multe lucruri si daca folosim mai mutle obiecte de acest tip, e normal ca va dura mai mult.

Testare la nivel de integrare

Testarea la nivel de integrare este faza in testarea software in care modulele software individuale sunt combinate si testate impreuna, ca un grup. Testare la nivel de integrare este efectuata pentru a evalua conformitatea unui sistem sau componenta cu cerinte functionale specificate.

Avantajul testarii la nivel de integrare este ca putem sa verificam functionarea unor componente integrate intre ele si cu sisteme externe, cum ar fi baza de date. In cazul dependentelor externe, exista doua categorii: dependente gestionate si dependente negestionate. Dependentele gestionate sunt dependente care sunt accesibile doar prin aplicatie, cum ar fi baza de date. Dependentele negestionate sunt dependente externe asupra carora nu avem control, cum ar fi o magistrala de mesaje.

Aceasta aplicatie este construita folosind Spring Boot, de aceea pentru testarea la nivel de integrare putem sa folosim tool-ul de testare integrat in Spring Boot, numit SpringBootTest. SpringBootTest foloseste in spate JUnit. Fiindca aceasta aplicatie este partea de backend unei aplicatii web mai complexe, foloseste servicii REST. In cadrul testelor la nivel de integrate au fost testate Controllere si prin acesta practic testam un flow, de la accesarea bazei de date pana la trimiterea unui raspuns la request-urile de tip GET si POST.

Ca sa generez niste request-uri de tip GET si POST, am folosit clasa MockMvc, care este implementata in SpringBootTest, aceasta este foarte importanta, deoarece in acest fel pot sa verific daca metodele dintr-un Controller raspund corect la request-urile de tip GET/POST. Ca si baza de date, am folosit o baza de date Postgres, aplicatia are acces la date folosind cateva clase de tip Repository.

In cadrul testelor la nivel de integrare, realizate in acest proiect, sunt verificate urmatoarele:

- gasirea contului dupa nume de utilizator si parola
- adaugarea unui utilizator de tip doctor in baza de date
- gasirea lista pacientilor
- gasirea utilizatorului de tip doctor dupa ID de cont
- stergerea unui pacient
- gasirea unui pacient dupa ID

Din proiectul prezentat mai sus, au fost testate 2 clase, clasa AccountController si DoctorController. Cele doua clase sunt de tip Controller, fiecare comunica cu baza de date folosind niste clase de tip Service, care folosesc niste clase de tip Repository.

Clasa AccountControllerTest:

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class AccountControllerTest {

    @Autowired
    private WebApplicationContext webApplicationContext;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }

    @Autowired
    private AccountRepository accountRepository;

    @Test
    public void getAccountByCredentials_WithOKResult() throws Exception {
        Account account = new Account("accountUserName", "accountPassword1234", AccountType.PATIENT);
        accountRepository.save(account);

        mockMvc.perform(MockMvcRequestBuilders
                .get("/account/loginCredentials/{user}/{password}", account.getUserName(), account.getPassword())
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.userName").value("accountUserName"))
                .andExpect(MockMvcResultMatchers.jsonPath("$.password").value("accountPassword1234"))
                .andExpect(MockMvcResultMatchers.jsonPath("$.accountType").value("PATIENT"));
    }

    @Test
    public void getAccountByCredentials_WithBADResult() throws Exception {

        mockMvc.perform(MockMvcRequestBuilders
                .get("/account/loginCredentials/{user}/{password}", "wrongUsername", "wrongPassword")
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isNotFound());
    }
}
```

Codul din clasa AccountController pe care au fost aplicate testele:

```
@RestController
@CrossOrigin
@RequestMapping(value = "/account")
public class AccountController {
```

```
    private final AccountService accountService;

    @Autowired
    public AccountController(AccountService accountService) {
        this.accountService = accountService;
    }

    @GetMapping(value = "/loginCredentials/{user}/{id}")
    public ResponseEntity<AccountDTO> getAccountByCredentials(@PathVariable("user") String username,
@PathVariable("id") String password) {
        AccountDTO dto = accountService.findAccountByUsernameAndPassword(username, password);
        return new ResponseEntity<>(dto, HttpStatus.OK);
    }
}
```

Clasa DoctorControllerTest:

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class DoctorControllerTest {

    @Autowired
    private WebApplicationContext webApplicationContext;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }

    @Autowired
    private AccountRepository accountRepository;

    @Autowired
    private DoctorRepository doctorRepository;

    @Autowired
    private PatientRepository patientRepository;

    public static String asJsonString(final Object obj) {
        try {
            System.out.println(new ObjectMapper().writeValueAsString(obj));
            return new ObjectMapper().writeValueAsString(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Test
    public void insertDoctorTest() throws Exception {
        Account drAccount = new Account("doctorUser", "passDoctor1234", AccountType.DOCTOR);
        DoctorDTO doctorDTO = new DoctorDTO(drAccount);

        mockMvc.perform(MockMvcRequestBuilders
                .post("/doctor/insert")
                .content(asJsonString(doctorDTO))
                .contentType(MediaType.APPLICATION_JSON)
```

```
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isCreated())
                .andExpect(MockMvcResultMatchers.forwardedUrl(null))
                .andExpect(MockMvcResultMatchers.redirectedUrl(null));
    }

    @Test
    public void getPatientsTest() throws Exception {
        Account account1 = new Account("patient1User", "patient1Pass", AccountType.PATIENT);
        Account account2 = new Account("patient2User", "patient2Pass", AccountType.PATIENT);
        Patient patient1 = new Patient("Patient1", LocalDate.of(1998, 6, 15), "Male", "Cluj-Napoca", "Headache", null,
account1);
        Patient patient2 = new Patient("Patient2", LocalDate.of(1996, 12, 21), "Female", "Oradea", "Stomachache", null,
account2);

        patientRepository.deleteAll();
        accountRepository.save(account1);
        accountRepository.save(account2);
        patientRepository.save(patient1);
        patientRepository.save(patient2);

        mockMvc.perform(MockMvcRequestBuilders
                .get("/doctor/patients")
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$[0].name").value("Patient1"))
                .andExpect(MockMvcResultMatchers.jsonPath("$[1].name").value("Patient2"))
                .andExpect(MockMvcResultMatchers.jsonPath("$[0].gender").value("Male"))
                .andExpect(MockMvcResultMatchers.jsonPath("$[1].gender").value("Female"));
    }

    @Test
    public void getDoctorByAccountIDTest_WithOKResult() throws Exception {
        Account drAccount = new Account("doctorUserID", "passIDDoctor1234", AccountType.DOCTOR);
        Doctor doctor = new Doctor(drAccount);

        accountRepository.save(drAccount);
        doctorRepository.save(doctor);

        Doctor savedDR = doctorRepository.findDoctorByAccount_UserName(drAccount.getUserName()).get();
        Assert.assertNotEquals(savedDR, null);

        mockMvc.perform(MockMvcRequestBuilders
                .get("/doctor/account/{accountID}", savedDR.getAccount().getId())
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.id").value(savedDR.getId().toString()));
    }

    @Test
    public void getDoctorByAccountIDTest_WithBADResult() throws Exception {
        UUID wrongAccountID = UUID.randomUUID();

        mockMvc.perform(MockMvcRequestBuilders
                .get("/doctor/account/{accountID}", wrongAccountID)
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
```

```
                .andExpect(MockMvcResultMatchers.status().isNotFound());
    }


    @Test
    public void deletePatientTest() throws Exception {
        Account patientAccount = new Account("patientDeleteUser", "patientDeletePass", AccountType.PATIENT);
        Patient deletePatient = new Patient("PatientDelete", LocalDate.of(1991, 4, 2), "Male", "Cluj-Napoca", "Headache",
null, patientAccount);

        accountRepository.save(patientAccount);
        patientRepository.save(deletePatient);

        Patient savedPatient = patientRepository.findPatientByName(deletePatient.getName()).get();
        Assert.assertNotEquals(savedPatient, null);

        mockMvc.perform(MockMvcRequestBuilders
                .post("/doctor/patient/delete/{id}", savedPatient.getId())
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isOk());


        Assert.assertThrows(NoSuchElementException.class, () -> patientRepository.findById(savedPatient.getId()).get());
    }

    @Test
    public void getPatientTest() throws Exception {
        Account patientAccount = new Account("patientGetUser", "patientGEtPass", AccountType.PATIENT);
        Patient getPatient = new Patient("PatientGet", LocalDate.of(1990, 9, 2), "Male", "Cluj-Napoca", "Headache", null,
patientAccount);

        accountRepository.save(patientAccount);
        patientRepository.save(getPatient);

        Patient savedPatient = patientRepository.findPatientByName(getPatient.getName()).get();
        Assert.assertNotEquals(savedPatient, null);

        mockMvc.perform(MockMvcRequestBuilders
                .get("/doctor/patient/{id}", savedPatient.getId())
                .accept(MediaType.APPLICATION_JSON))
                .andDo(MockMvcResultHandlers.print())
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.id").value(savedPatient.getId().toString()))
                .andExpect(MockMvcResultMatchers.jsonPath("$.name").value(savedPatient.getName()));
    }
}
```

Codul din clasa DoctorController pe care au fost aplicate testele:

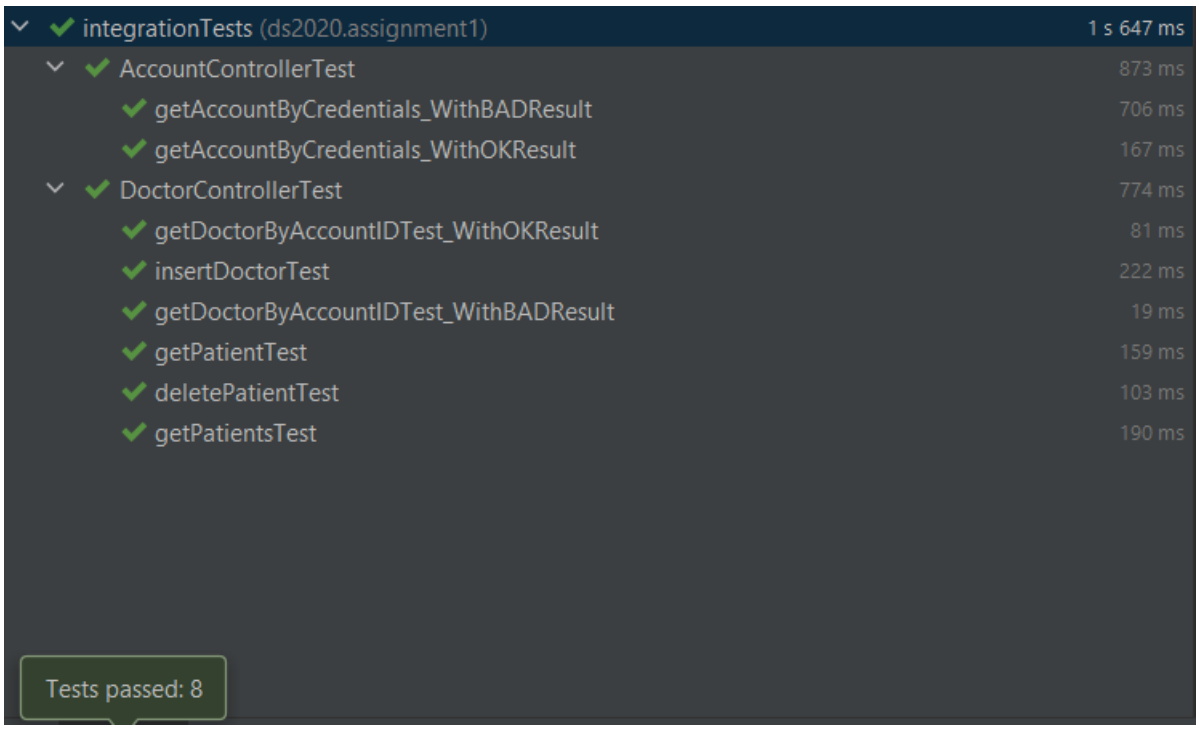
```
@RestController
@CrossOrigin
@RequestMapping(value = "/doctor")
public class DoctorController {

    private final DoctorService doctorService;
    private final PatientService patientService;
```

```
    private final CaregiverService caregiverService;
    private final MedicationService medicationService;
    private final MedicationPlanService medicationPlanService;

    @Autowired
    public DoctorController(PatientService patientService, CaregiverService caregiverService, MedicationService
medicationService, MedicationPlanService medicationPlanService, DoctorService doctorService) {
        this.patientService = patientService;
        this.caregiverService = caregiverService;
        this.medicationService = medicationService;
        this.medicationPlanService = medicationPlanService;
        this.doctorService = doctorService;
    }

    @PostMapping(value = "/insert")
    public ResponseEntity<UUID> insertDoctor(@Valid @RequestBody DoctorDTO doctorDTO) {
        doctorDTO.getAccount().setAccountType(AccountType.DOCTOR);
        UUID doctorID = doctorService.insertDoctor(doctorDTO);
        return new ResponseEntity<>(doctorID, HttpStatus.CREATED);
    }

    @GetMapping(value = "/patients")
    public ResponseEntity<List<PatientDTO>> getPatients() {
        List<PatientDTO> dtos = patientService.findPatients();
        for (PatientDTO dto : dtos) {
            Link patientLink = linkTo(methodOn(PatientController.class)
                    .getPatient(dto.getId())).withRel("patientDetails");
            dto.add(patientLink);
        }
        return new ResponseEntity<>(dtos, HttpStatus.OK);
    }

@GetMapping(value = "/account/{accountid}")
public ResponseEntity<DoctorDTO> getDoctorByAccountID(@PathVariable("accountid") UUID accountId) {
    DoctorDTO dto = doctorService.findDoctorByAccountId(accountId);
    return new ResponseEntity<>(dto, HttpStatus.OK);
}
@PostMapping(value = "/patient/delete/{id}")
public ResponseEntity<UUID> deletePatient(@PathVariable("id") UUID patientId) {
    patientService.deletePatient(patientId);
    return new ResponseEntity<>(patientId, HttpStatus.OK);
}

@GetMapping(value = "/patient/{id}")
public ResponseEntity<PatientDTO> getPatient(@PathVariable("id") UUID patientId) {
    PatientDTO dto = patientService.findPatientById(patientId);
    return new ResponseEntity<>(dto, HttpStatus.OK);
}
}
```

Pentru verificarea si evaluarea testelor am folosit clasa Assert din framework-ul JUnit si metoda andExpect() din clasa MockMvc. Metoda *Assert.assertNotEquals(valoare asteptata, valoare primita)* verifica daca cei doi parametri nu sunt egale. O alta metoda, tot din framework-ul Assert, a fost folosit in anumite cazuri (unde am verificat daca metoda arunca o exceptie, in cazul unei date gresite): *Assert.assertThrows(tipul exceptiei, metoda care arunca o exceptie)*. In acest fel a fost testat fiecare ramura dintr-o metoda: si cand are un rezultat valid, si cand arunca o exceptie.

Un alt tip de verificare folosita este metoda *andExpect()*. Aceasta metoda poate fi folosit la mai multe lucruri: am verificat daca status-ul de HTTP a rezultatului este corect sau daca body-ul raspunsului la un request de GET/POST contine o anumita valoare.

Rezultatele testarilor la nivel de integrare sunt prezentate mai jos:



Datele introduse (conturile create pentru fiecare utilizator introdus) in baza de date:

	id [PK] bytea	account_type integer	password character varying (255)	user_name character varying (255)
1	[binary data]	2	accountPassword1234	accountUserName
2	[binary data]	1	passIDDoctor1234	doctorUserID
3	[binary data]	1	passDoctor1234	doctorUser
4	[binary data]	2	patientGEtPass	patientGetUser
5	[binary data]	2	patient1Pass	patient1User
6	[binary data]	2	patient2Pass	patient2User

In cazul fiecarui test au fost generate niste date de test, care au fost introduse in baza de date Postgres, deci pentru fiecare metoda de test am accesat baza de date conectata la aceasta aplicatie. Fiecare test a rulat cu succes si in baza de date apar datele introduse, dupa cum se vede pe imaginea de mai sus.

Aceste teste au fost foarte utile, am reusit sa testez o buna parte din aplicatie de backend, si din fericire toate testele au trecut, nu am gasit nicio eroare de implementare si nu ar trebuit sa

refactorizez codul sursa. Aceste teste vor fi folositoare si in viitor, daca o sa mai adaug niste functionalitati sau daca o sa mai schimb logica la cateva metode.

Testare la nivel de sistem

Pentru testarea la nivel de sistem am ales testarea end-to-end. Testarea end-to-end (E2E) este o metoda de testare software care valideaza intregul sistem de la inceput pana la sfarsit, impreuna cu integrarea acestuia cu interfete externe. Scopul testarii end-to-end este testarea intregului sistem pentru dependente, integritatea datelor si comunicarea cu alte sisteme, interfete si baze de date pentru a exercita o scena de productie completa.

In acest fel, am reusit sa testez daca backend-ul functioneaza corect impreuna cu frontend-ul scris in React si cu baza de date Postgres. Practic, cu acest tip de testare, m-am asigurat ca flowul aplicatiei (de la baza de date pana la frontend) este unul corect si ca acest sistem functioneaza asa cum mi-a dorit eu.

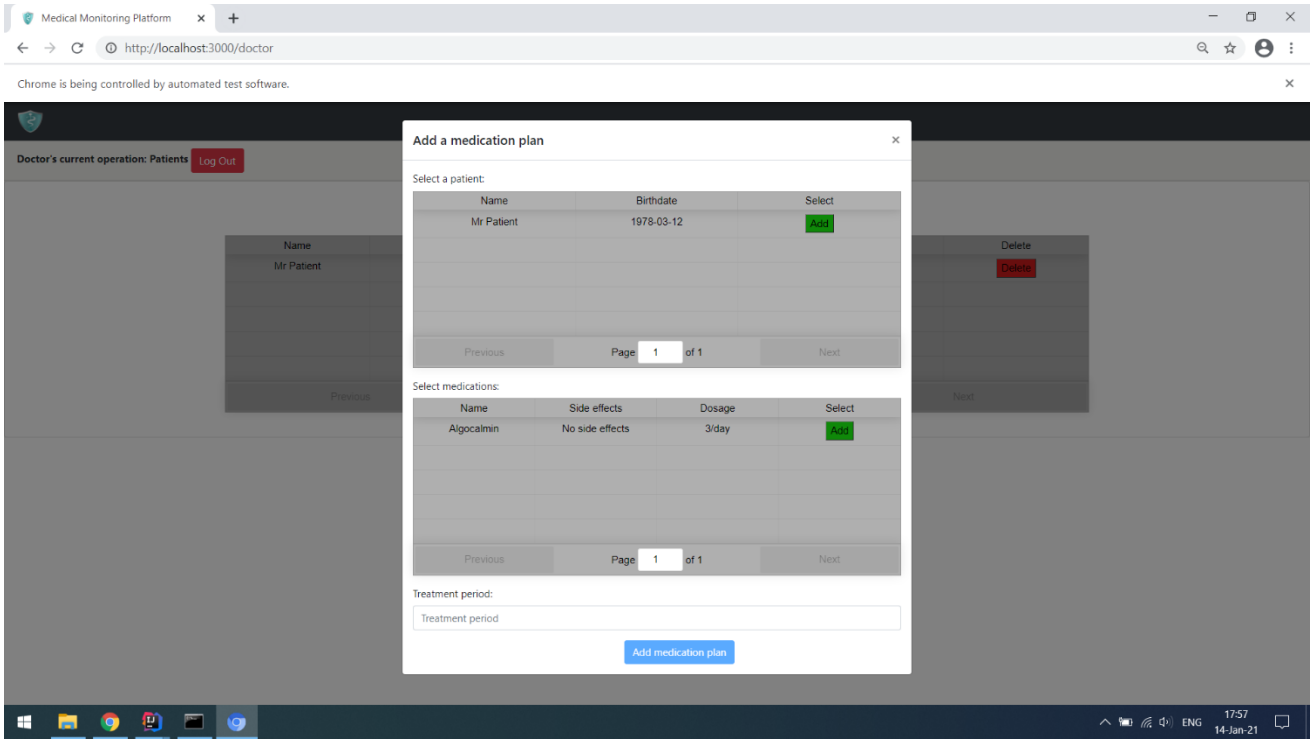
Testarea end-to-end am realizat cu ajutorul librariei Taiko. Aceasta este o biblioteca Node.js open-source si gratuit, care este folosit pentru testarea aplicatiilor web. Taiko are un API foarte simplu, scriind putin cod ajungem sa testam o mare parte din aplicatia noastra. Testele Taiko sunt scrise in JavaScript sau in orice alt limbaj care se compileaza in JavaScript. Taiko trateaza browserul ca si un blackbox si putem sa scriem scripturi concentrand pe pagina web, fara a inspecta codul sursa al acesteia.

In cadrul testului la nivel de sistem, am verificat daca un utilizator de tip doctor poate sa creeze un plan de medicamente pentru un pacient existent in baza de date. Practic, prin aceasta testare, am verificat mai multe clase si componente atat pe backend, cat si pe frontend: sunt verificate clasele de tip Repository, Service si Controller pentru utilizatorul de tip doctor si este testat pagina web si functionalitatile de pe pagina web unui utilizator de tip doctor.

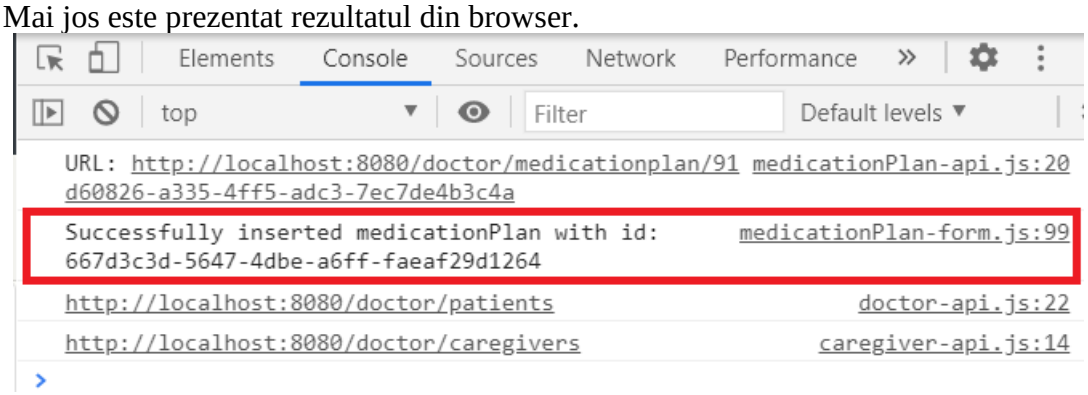
Scriptul de test:

```
const { openBrowser, goto, click, below, textBox, into, write, closeBrowser } = require('taiko');
(async () => {
    try {
        await openBrowser();
        await goto("http://localhost:3000/doctor");
        await click("Add medication plan");
        await click("Add");
        await click("Add");
        await write("14",into(textBox(below("Treatment period:"))));
        await click("Add medication plan");
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

Acest script prima data deschide browserul, dupa care acceseaza pagina unui utilizator de tip doctor, care este rulat folosind nginx pe localhost si pe portul 3000. Cand se incarca pagina, face un click pe butonul „Add medication plan” dupa care deschide un pop-up window, unde selecteaza pacientul si medicamentul apasand pe buton „Add”.



Dupa ce a selectat pacientul si medicamentul, in textboxul de „Treatment period” introduce 14, care se refera la numarul de zile, iar la final apasa pe butonul „Add medication plan”. La final inchide browserul deschis pentru testare. Rezultatul putem sa verificam si in browser, dar si in baza de date. Mai jos este prezentat rezultatul din browser.



Rezultatul testului la nivel de sistem este prezentat mai jos:

```
Your environment has been set up for using Node.js 14.15.0 (x64) and npm.

C:\Users\Tamas>npx taiko endToEndTest.js
[PASS] Browser opened
[PASS] Navigated to URL http://localhost:3000/doctor
[PASS] Clicked element matching text "Add medication plan"  1 times
[PASS] Clicked element matching text "Add"  1 times
[PASS] Clicked element matching text "Add"  1 times
[PASS] Wrote 14 into the textBox below Treatment period:
[PASS] Clicked element matching text "Add medication plan"  1 times
[PASS] Browser closed
```

Rezultatul arata ca Taiko s-a terminat fiecare operatie cu succes. Rezultatul generat de Taiko si rezultatul din browser arata ca testul la nivel de sistem a rulat cu succes, functioneaza atat comunicarea intre baza de date si aplicatia backend, cat si comunicarea intre aplicatia backend si aplicatia frontend.

Testul de tip end-to-end a fost foarte folositor, in acest fel am reusit sa testez toate componenetele acestui proiect.A fost cel mai complex test, care testeaza toate componenetele deodata, din fericire testul a trecut si nu am gasit nicio eroare de implementare si nu ar trebuit sa refactorizez codul sursa ca sa rulez acest tip de test, ar trebuit doar sa instalez biblioteca Taiko.

Concluzii

In concluzie, pot spune ca aceste teste au inbunatatit proiectul meu, deoarece in acest fel sunt mai sigur ca proiectul va functiona si in situatii unice, mai complexe si ca proiectul meu a fost implementat corect. Pe langa acesta, lucrand la acest proiect mi-am imbogatatit cunostintele, fiindca am invatat sa folosesc niste framework-uri de testare pe care pot sa folosesc si in viitor.

Un alt lucru important este ca in acest fel am inteles cat de important este sa testam functionalitatile proiectului si ca este mult mai eficient scrierea codului daca scriem si teste in paralel. Intr-adevar dureaza mai mult daca scriem si teste, dar este greu pana cand ne obisnuim cu fiecare framework, dupa care scrierea testelor devine un proces simplu, care merita efortul , deoarece putem sa scriem si sa testam codul in acelasi timp si daca gasim o functionalitate incorecta, putem sa rezolvam mult mai repede, inainte de a avea dependinte cu acest fragment de cod.