SEM Assignment 3 - group 33A

Laimonas Lipinskas (5559375)
Lotte Kremer (4861957)
Nicolae Radu (5527104)
Razvan Loghin (5480620)
Dawid Plona (5205018)
Vlad Nitu (5529557)

January 25, 2023

Task 1: Automated Mutation Testing

In the following sections the mutation scores of the classes we changed are displayed. Per class there is a link to the commit and a screenshot of the scores before and after the change.

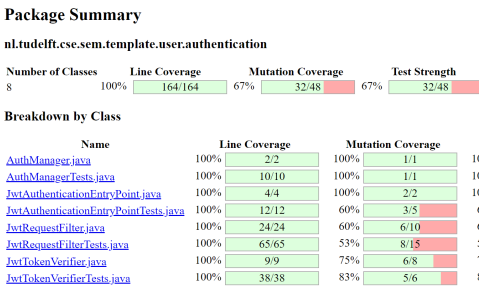
JwtRequestFilter in the user microservice

Commits

- c52440b7, 8aa054d6 and 8c9a32b8 were the commits that raised the mutation score from 60% to 80%. In the last of those commits we made a small change to make the test more solid.
- 628eef04 was the final commit to make the mutation score 90% by altering one more test
- *... we could place the link to the MR here?*

Before and after screenshots

In Figure 1 it can be seen that the mutation score of the JwtRequestFilter class in the User microservice was 60% due to 4 mutants that survived. By adding new assertions to the test cases this improved to 90%, as seen in Figure 2.



Package Summary

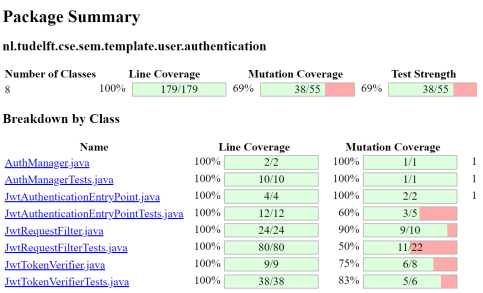
nl.tudelft.cse.sem.template.user.authentication

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	100%	164/164	67%	32/48	67%	32/48

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AuthManager.java	100%	2/2	100%	1/1
AuthManagerTests.java	100%	10/10	100%	1/1
JwtAuthenticationEntryPoint.java	100%	4/4	100%	2/2
JwtAuthenticationEntryPointTests.java	100%	12/12	60%	3/5
JwtRequestFilter.java	100%	24/24	60%	6/10
JwtRequestFilterTests.java	100%	65/65	53%	8/15
JwtTokenVerifier.java	100%	9/9	75%	6/8
JwtTokenVerifierTests.java	100%	38/38	83%	5/6

Figure 1: Pitest report before new test cases



Package Summary

nl.tudelft.cse.sem.template.user.authentication

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	100%	179/179	69%	38/55	69%	38/55

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AuthManager.java	100%	2/2	100%	1/1
AuthManagerTests.java	100%	10/10	100%	1/1
JwtAuthenticationEntryPoint.java	100%	4/4	100%	2/2
JwtAuthenticationEntryPointTests.java	100%	12/12	60%	3/5
JwtRequestFilter.java	100%	24/24	90%	9/10
JwtRequestFilterTests.java	100%	80/80	50%	11/22
JwtTokenVerifier.java	100%	9/9	75%	6/8
JwtTokenVerifierTests.java	100%	38/38	83%	5/6

Figure 2: Pitest report after new test cases

Matching microservice classes - Motivation

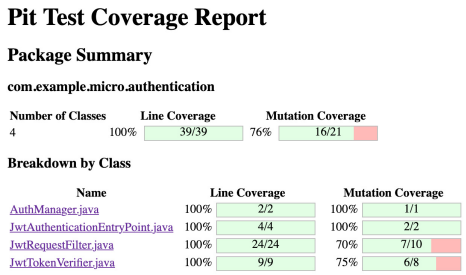
There were only two classes that had a method with mutation score $\leq 70\%$, $JwtRequestFilter$ and $NotificationPublisher$. Both had a surviving "removed call" type of mutant, which basically meant that we did not test whether that method was invoked (with the expected parameters). Thus, we had to validate whether the logs were the ones expected, both for $System.out$ and $System.err$ logs.

JwtRequestFilter in the Matching microservice

Commits

- 5f29d651, prepared the boilerplate code for capturing the logs produced by $System.err$ that we then used in 91aa3763 in order to raise the mutation score from 70% to 100%. In the last of those commits we made a small change to make the test more solid. Thus, 91aa3763 is the final commit in which we've written the test that kills the mutant.

We solved this issue by instantiating an 'outputStreamCaptor' entity that allowed us to "spy" over what was logged through 'System.out' calls. Thus, we were able to asses whether the correct exception message was being thrown when we were "Unable to parse JWT token" or that the token expired.



Pit Test Coverage Report

Package Summary

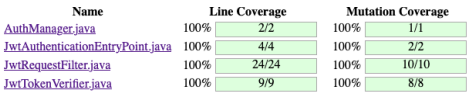
com.example.micro.authentication

Number of Classes	Line Coverage		Mutation Coverage	
4	100%	39/39	76%	16/21

Breakdown by Class

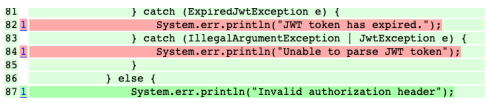
Name	Line Coverage		Mutation Coverage	
AuthManager.java	100%	2/2	100%	1/1
JwtAuthenticationEntryPoint.java	100%	4/4	100%	2/2
JwtRequestFilter.java	100%	24/24	70%	7/10
JwtTokenVerifier.java	100%	9/9	75%	6/8

Figure 3: PITest report BEFORE



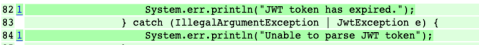
Name	Line Coverage		Mutation Coverage	
AuthManager.java	100%	2/2	100%	1/1
JwtAuthenticationEntryPoint.java	100%	4/4	100%	2/2
JwtRequestFilter.java	100%	24/24	100%	10/10
JwtTokenVerifier.java	100%	9/9	100%	8/8

Figure 4: PITest report AFTER



```
81                  } catch (ExpiredJwtException e) {
82 1                    System.err.println("JWT token has expired.");
83                  } catch (IllegalArgumentException | JwtException e) {
84 1                    System.err.println("Unable to parse JWT token");
85                  }
86              } else {
87 1                System.err.println("Invalid authorization header");
```

Figure 5: Mutants (2) that survived before



```
82 1                    System.err.println("JWT token has expired.");
83                  } catch (IllegalArgumentException | JwtException e) {
84 1                    System.err.println("Unable to parse JWT token");
```

Figure 6: Both mutants were killed after

ActivityService in the Actiivty microservice

Commits

- Previously the $ActivityService$ class had a mutation score of 68% as shown in Fig. 7, because of multiple mutants that survived the original test suite. The specific mutants can be seen in Fig. 9. After the commit 726aafb1 that implements all the JUnit tests

necessary to kill the muntants and a68c2e66 that fixes the pipeline, we obtained the follwing results in the PITest report shown in Fig. 8.

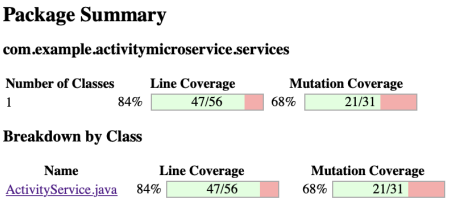


Figure 7: PITest report BEFORE

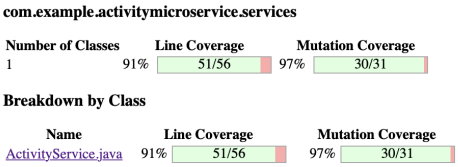


Figure 8: PITest report AFTER

Figure 9: Mutants that survived before

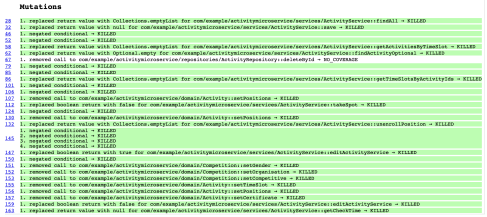


Figure 10: Mutants were killed after

NotificationPublisher in the Matching microservice

Commits

- Previously, *notifyUser* method from *NotificationPublisher* had 0% mutation coverage, as there was only one mutant that existed, which was not killed before. Thus, in c1061119, we improve the mutation score from 0% to 100% (by killing 1/1 mutants through one JUnit test, verifying that *System.out* was invoked, and also whether the logs are indeed the message of the error that we expect)

We solved this issue by following a similar procedure to the mutation killing from *JwtRequestFilter*, but now we wanted to "spy" the *System.out* logs, instead of the *System.err* ones, so further adjustments were needed

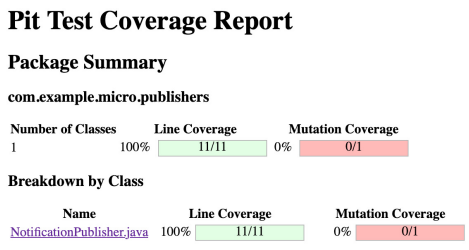


Figure 11: PITest report BEFORE

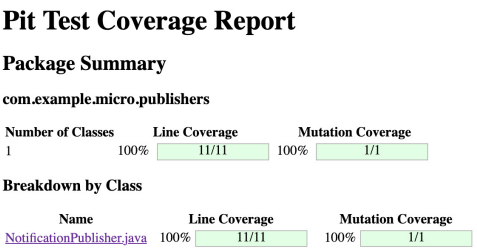


Figure 12: PITest report AFTER

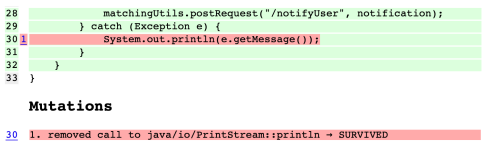


Figure 13: Mutant that survived before

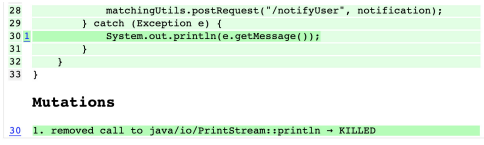


Figure 14: Mutant killed afterward

NotificationDatabaseService in the Notification microservice

Commits

- 62dfb4fb

Mutation score before was 25% because the tests were not checking the return value, so decided to just rewrite/bolster the tests.

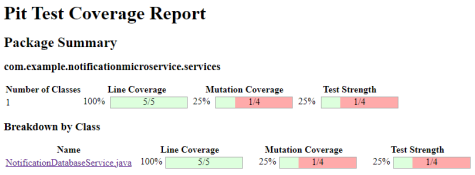


Figure 15: PITest report BEFORE

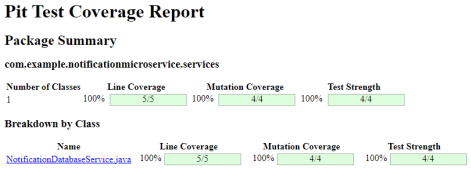


Figure 16: PITest report AFTER

Task 2: Manual Mutation Testing

Core domain

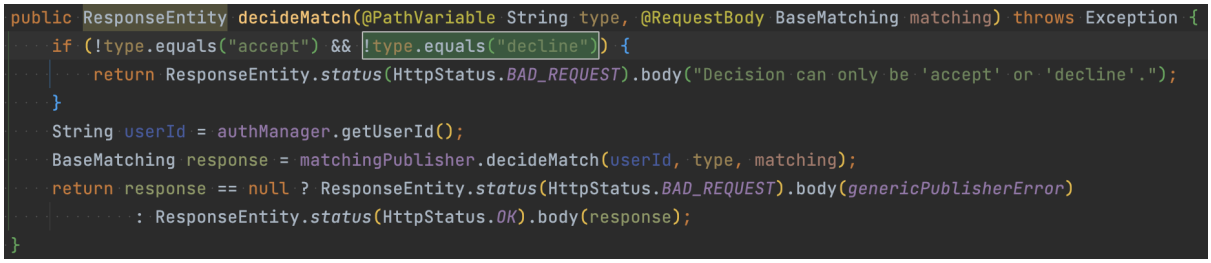
For solving the second task of this assignment, we decided to choose *users* as the core domain that we want to create the manual mutation tests for, as this one was the most complex and definitely one of the most relevant for our entire system, as our matchmaking app is centered around users, having as its main purpose finding suitable teammates (and activities) for rowers.

Critical classes (4)

We chose to manually create mutants for the following four critical classes within the $User$ domain:

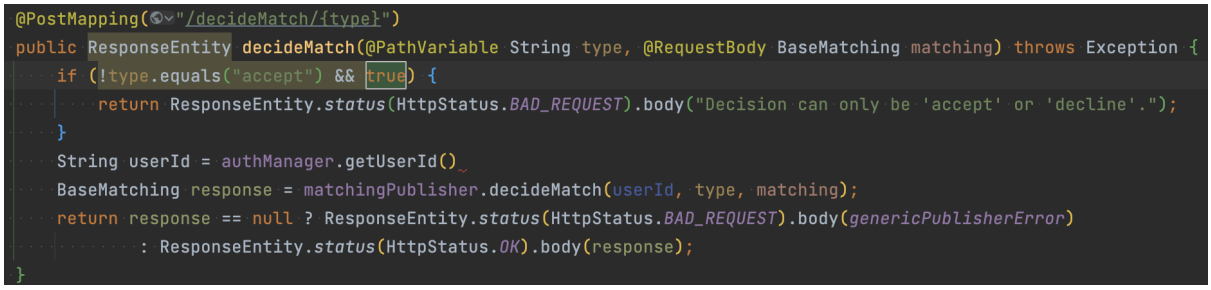
1. **UserMatchingController**, as this one ensures the communication between the User subsystem and the Matching one is always satisfied. We consider this class crucial as rowers cannot be assigned to an activity if the matching process does not take place. As the users always initiate their desire to be allocated a position in a specific activity, implementing such a "Controller" that allows subsystems to communicate through HTTP requests via API calls is mandatory. Therefore, making sure that this component behaves as desired became one of our priorities, and by adding manual mutation tests we can increase the chances of this sub-component being free of bugs. That being said, we came up with two different manual mutants for this specific class:

- a.) **Remove conditionals** type of mutant: we found that this mutant was alive after running our test-suite in the $decideMatch$ method. If you want to find more about this mutant, you can further read and getting more insight about how this mutant works, we recommend checking Remove Conditionals Mutator (REMOVE_CONDITIONALS) section. In short, what this mutant does is: it replaces one of the conditions from the if statement to "true" (the second one in our case). Here is a quick visualization:



```
public ResponseEntity decideMatch(@PathVariable String type, @RequestBody BaseMatching matching) throws Exception {
    if (!type.equals("accept") && !type.equals("decline")) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Decision can only be 'accept' or 'decline'.");
    }
    String userId = authManager.getUserId();
    BaseMatching response = matchingPublisher.decideMatch(userId, type, matching);
    return response == null ? ResponseEntity.status(HttpStatus.BAD_REQUEST).body(genericPublisherError)
            : ResponseEntity.status(HttpStatus.OK).body(response);
}
```

Figure 17: Correct code (before introducing mutant)



```
@PostMapping("/decideMatch/{type}")
public ResponseEntity decideMatch(@PathVariable String type, @RequestBody BaseMatching matching) throws Exception {
    if (!type.equals("accept") && true) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Decision can only be 'accept' or 'decline'.");
    }
    String userId = authManager.getUserId()
    BaseMatching response = matchingPublisher.decideMatch(userId, type, matching);
    return response == null ? ResponseEntity.status(HttpStatus.BAD_REQUEST).body(genericPublisherError)
            : ResponseEntity.status(HttpStatus.OK).body(response);
}
```

Figure 18: Altered code (after introducing mutant)

When using the previous test-suite, all tests passed when using both the correct code and the mutated one (when we changed '!type.equals("decline")' into 'true', as depicted in figures 17 and 18). After adding the test referenced in commit 6a52840f, we managed to kill the mutant (as now the test-suite passes only for the correct code, while for the mutated one this new added test fails)

- b.) **Remove method call** type of mutant: - In the *chooseActivity* method, we observed that there was no test which exercised whether the matching activity had its *userId* changed after an activity was chosen (as the (user-matching) activity should be uniquely defined by a composite key consisting of both *userId* and *activityId*, so whenever we create such a match, the *userId* of the Matching entity has to be modified):

```
@PostMapping("/chooseActivity")
public ResponseEntity chooseActivity(@RequestBody BaseMatching matching) throws Exception {
    String userId = authManager.getUserId();
    matching.setUserId(userId);
    BaseMatching response = matchingPublisher.chooseActivity(matching);
    return response == null ? ResponseEntity.status(HttpStatus.BAD_REQUEST).body(genericPublisherError)
            : ResponseEntity.status(HttpStatus.OK).body(response);
}
```

Figure 19: Modify the userId of a matching

Thus, we tried removing this method call, and the entire suit was still passing. This indicated that the mutant we've just introduced into our code was surviving. As a follow-up, we've added a JUnit test in order to check whether, after choosing an activity, the matching's userId gets changed. The test can be found in the commit 563c1810 (which, indeed, passes on our previous correct code, but fails on the mutated / bugged code, the one in which we removed the method call)

2. **UserController**, as this class contains multiple API endpoints necessary for creating profiles and handling users' details. It is to be considered crucial, since no user profile might be successfully created or edited without utilising its contained methods by making HTTP requests through the API endpoints. Hence, thoroughly testing this class with use of various tools and approaches appears to be indispensable for ensuring desirable and bug-free experience for the end-users of the system. The manual mutant inserted and dealt with for this class is **remove method call**. In the *changeTimeSlots* method test suite there was no validation as to whether the time slots were actually altered to conform the request body contents. Therefore, removing the call to *setTimeSlots* did not affect the test suite result, at the same time rendering the method faulty, as shown in 20 and 21:

```
@PostMapping("/changeTimeSlot/{userId}")
public ResponseEntity changeTimeSlots(@PathVariable String userId,
                                      @RequestBody @NotNull Set<TimeSlot> timeslots) {
    if (!userId.equals(authManager.getUserId())) {
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body("The provided userId does not match your userId");
    }
    Optional<User> foundUser = userService.findUserById(userId);
    if (foundUser.isEmpty()) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("The user does not exist");
    }
    //foundUser.get().setTimeSlots(timeslots);
    return ResponseEntity.ok(userService.save(foundUser.get()));
}
```

Figure 20: Introduced mutant

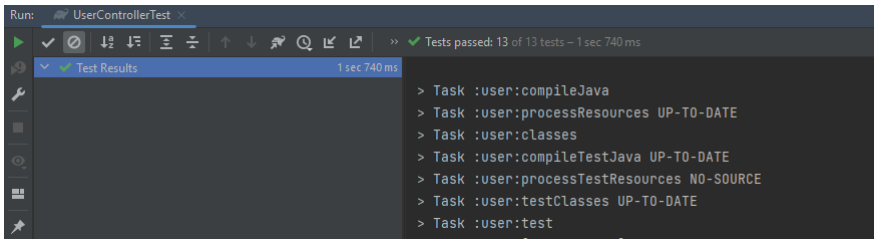


Figure 21: Test suite passing despite the mutant

To kill the surviving mutant, another JUnit test - *changeTimeSlotSetTimeslots* - was implemented specifically for this purpose in c91898cd, see 22:

```
@Test
void changeTimeSlotSetTimeslots() throws Exception {
    when(authManager.getUserId()).thenReturn( value: "userId");
    TimeSlot timeSlot = new TimeSlot(LocalDateTime.of( year: 2004,  month: 12,  dayOfMonth: 1,  hour: 23,  minute: 15),
            LocalDateTime.of( year: 2004,  month: 12,  dayOfMonth: 1,  hour: 23,  minute: 15));
    Set<TimeSlot> times = Set.of(timeSlot);
    when(userService.findUserById( userId: "userId")).thenReturn(Optional.of(user));
    when(userService.save(user)).thenReturn(user);
    MvcResult mvcResult = mockMvc
            .perform(post( urlTemplate: "/changeTimeSlot/userId")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(times))
            )
            .andExpect(status().isOk())
            .andReturn();
    String contentAsString = mvcResult.getResponse().getContentAsString();
    verify(userService).save(any());
    User obtained = objectMapper.readValue(contentAsString, User.class);
    Set<TimeSlot> changedTimes = obtained.getTimeSlots();
    assertEquals(changedTimes, times);
}
```

Figure 22: Test implemented to kill the mutant

The newly introduced test managed to kill the mutant, as show in 23

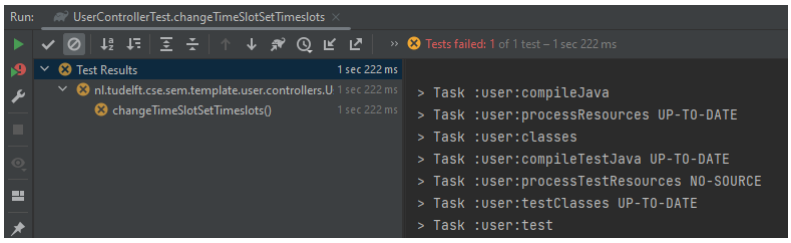


Figure 23: Test suite passing despite the mutant

Additionally, the original code without the manually inserted mutant still passes the updated test suite, see 24:

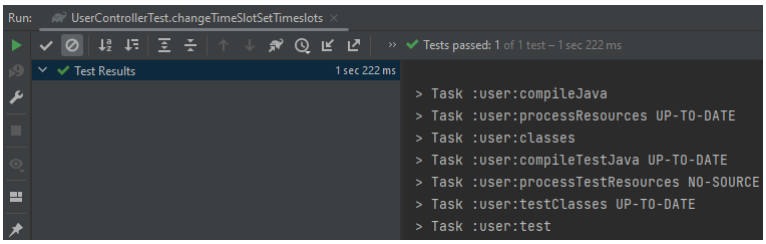


Figure 24: Test suite passing despite the mutant

3. **InputValidation**, as this class is responsible for checking if a user is creating an account with valid positions and a valid gender. If it deemes the user's positions/gender as invalid it will not allow to create the user. Thus we need to make sure that InputValidation works as expected for all positions/genders. The mutant we introduced was removing a valid position from the class's valid positions list as it would be easy to forget to add a new position. In the old test case nothing was making sure that all the valid positions should be valid and because of that the mutant survived but in the updated test case it was killed since we now account for that.

```
private static Set<String> validPositions = Set.of("cox",
        "coach",
        "port side rower",
        //"starboard side rower",
        "sculling rower");

private static final Set<Character> possibleGenders = Set.of('M', /*'m',*/ 'F', 'f');
```

Figure 25: Introduced mutant

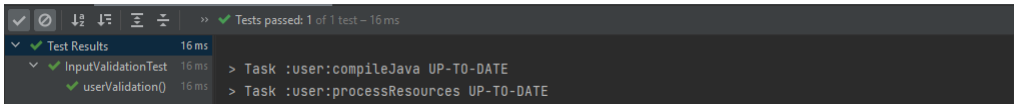


Figure 26: Test suite passing despite the mutant

To kill the surviving mutant, another JUnit test - *userValidation* - was updated 971b574c, see 27:

```java
@Test
void userValidation() {
    char genderValidM = 'M';
    Set<String> validPositions = Set.of("cox",
            "coach",
            "port side rower",
            "starboard side rower",
            "sculling rower");
    assertNull(InputValidation.validate(validPositions, genderValidM));

    Set<Character> validGenders = Set.of('M', 'm', 'F', 'f');
    for (Character gender : validGenders) {
        assertNull(InputValidation.validate(Set.of("coach"),  gender));
    }

    char genderInvalid = 'N';
    assertEquals(InputValidation.validate(validPositions, genderInvalid).getSecond(),  actual: "The provided gender is invalid!");
    Set<String> invalidPositions = Set.of("invalid");
    assertEquals(InputValidation.validate(invalidPositions, genderValidM).getSecond(),
            actual: "One of the positions that you provided is not valid!");
}
```

Figure 27: Test updated test managed to kill the mutant

The newly introduced test managed to kill the mutant, as show in 28

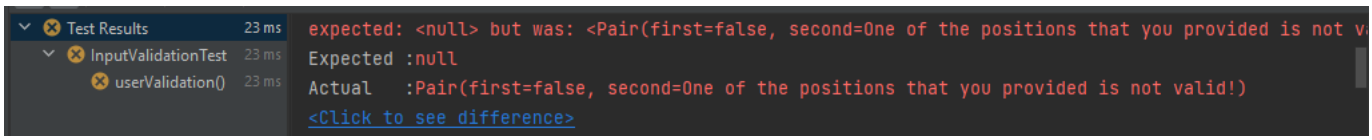


Figure 28: Test suite passing despite the mutant

4. **UserActivityController** is the class that ensures the communication between the User and Activity microservices. We consider this crucial as Users of the application would not be able to create or delete activities, making the whole purpose of the system useless. This is why we consider it essential that this class should be free of mutants and as comprehensively tested as it can be. The manual mutant that we choose to insert it the **remove method call**. In the *createActivity* method test suite there is no validation to whether the created activity is being completed with the right *type*. Therefore, removing the *activity.setType(type)* call does not affect the test suite result, as shown in pictures 29 and 30.

```java
@PostMapping("/createActivity/{type}")
public ResponseEntity createActivity(@Valid @RequestBody BaseActivity activity,
                                     @PathVariable String type) throws Exception {
    if (!validateType(type)) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Type can only be 'training' or 'competition'.");
    }
    if (!activity.getOwnerId().equals(authManager.getUserId())) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("The provided ownerId does not match your userId! Use "
                + authManager.getUserId() + " as the ownerId.");
    }
    //activity.setType(type);
    BaseActivity response = activityPublisher.createActivity(activity);
    return response == null ? ResponseEntity.status(HttpStatus.BAD_REQUEST).body(genericPublisherError)
            : ResponseEntity.status(HttpStatus.OK).body(response);
}
```

Figure 29: Introduced mutant

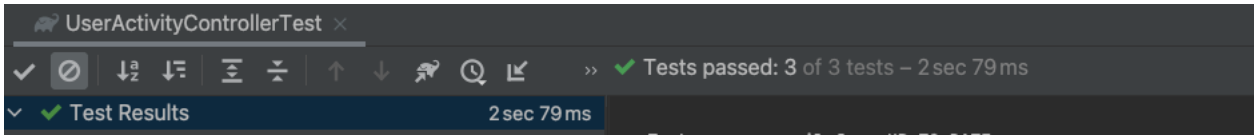


Figure 30: Test suite passing despite the mutant

To kill the mutant, another JUnit test - *createActivityCatchMutantTest* was created. In the first commit 8813eb6f, the created test was failing for the injected mutant, but also for the correct implementation of the method. The solution that was applied in the second commit 310c22ed, was to implement the equals function in the **BaseActivity** class, such that the moking would work. The final version of the *createActivityCatchMutantTest* can be seen in 31 and the effects can be seen in 32.

```
@Test
void createActivityCatchMutantTest() throws Exception {
    BaseActivity baseActivity = new BaseActivity();
    baseActivity.setOwnerId("valid");

    when(authManager.getUserId()).thenReturn("valid");
    // Initialise expectedBaseActivity object in order to force
    // `activityPublisher.createActivity` to return expected result
    // after setting the type of `baseActivity` (which was sent via HTTP Post request) to "training"
    BaseActivity expectedBaseActivity = new BaseActivity();
    expectedBaseActivity.setOwnerId("valid");
    expectedBaseActivity.setType("training");

    when(activityPublisher.createActivity(expectedBaseActivity)).thenReturn(expectedBaseActivity);


    MvcResult mvcResult = mockMvc
            .perform(post( urlTemplate: "/createActivity/training")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(baseActivity))
            )
            .andDo(MockMvcResultHandlers.print())
            .andExpect(status().isOk())
            .andReturn();
    String contentAsString = mvcResult.getResponse().getContentAsString();
    BaseActivity obtained = objectMapper.readValue(contentAsString, BaseActivity.class);
    assertThat(obtained.getOwnerId()).isEqualTo(baseActivity.getOwnerId());
    assertThat(obtained.getType()).isEqualTo("training");
}
```

Figure 31: Test implemented to kill the mutant

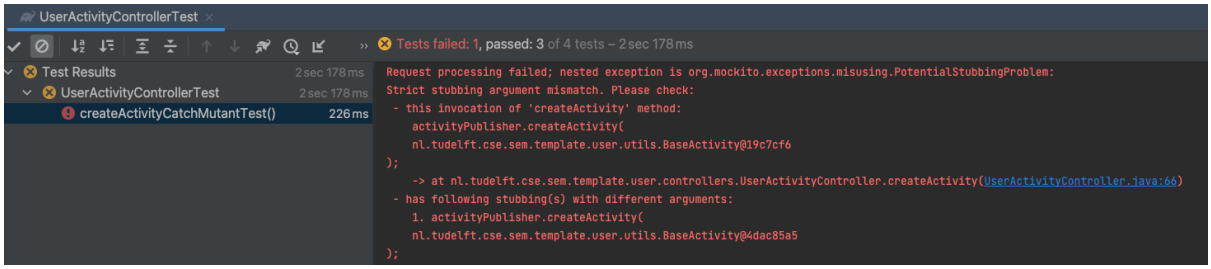


Figure 32: Test suite failing because of the mutant