ΕΜΕΊΣ

CMXM

CodusMaximus

Hyper Perform
Testing Document

Organisation: `https://github.com/Hyperperform`

Developers:
Claudio Da Silva *14205892*
Rohan Chhipa *14188377*
Avinash Singh *14043778*
Jason Gordon *14405025*

Updated October 11, 2016

Contents

1 Introduction

This document explains implementation of unit tests which were used to test each component in an isolated environment. It also includes an overview of the technologies used as well as instructions on how to execute these tests.

2 Purpose

The HyperPerform system was made to be able to track employee performance. One of the main goals of the system is to be able to use the system in any industrial or academic field. To ensure this each component of the system is decoupled from every other. All aspects of the system are implemented such that they conform to the given contracts. Unit testing is essential to ensure that these contracts are not violated in anyway and to ensure that the components that expose the services (REST wrapping) are all working properly before the system is deployed.

3 Technologies

Since the system was primarily coded with Java we used JUnit to carry out the unit tests for each of the components. We also used Spring to allow us to use dependency injection within each test, this allows us to easily inject mocks and test components in an isolated environment.

Maven was used as the build tool for this system. When building the system from its source code all unit tests are automatically executed by Maven.

Another technology which proved useful was a Mock Dispatcher framework which ships with RESTEasy. The mock dispatcher allows one to easily and efficiently test REST API's without having to deploy the component to an application server.

4 Testing environment

- **Programming Languages:** Java with JavaEE for the back-end server, front-end relies on AngularJS with Bootstrap.
- **Testing frameworks:** Unit testing is done using JUnit. This is used in conjunction with Spring, which is used to achieve dependency injection within the unit tests.

- **Operating System:** The unit tests are not platform specific. The only requirement in terms of operating systems is that Maven must be supported. If Maven is supported then all unit test dependencies will be downloaded and all the tests will be executed automatically.
- **Internet Browsers:** The front-end system supports the latest versions of *GoogleChrome* and *Firefox*. There is limited support for other browsers.

5 Execution of unit tests

The project was developed using Maven as a build tool. Unit tests Thus each unit test can be easily found within the *src/test* directory of the project. To run the unit tests simply run the following command on terminal:

```
mvn clean test
```

All the necessary dependencies for the project will be automatically downloaded. **Note:** This process of downloading all the project dependencies might consume large amounts of data and time.

6 Test items

At this point in the project the following features have been tested:

- GitListener which receives events from GitHub.
- GitPushEvent which is a POJO that contains information regarding a GitHub event.
- CalendarListener which receives events from Google Calendar service.
- CalendarMeeting which is a POJO that contains information of Meetings from Google Calendar.
- CalendarProject which is a POJO that contains information of Projects from Google Calendar.
- TravisListener which receives events from Travis CI.
- TravisEvent which is a POJO that contains information regarding a build triggered by a push event from GitHub
- AccessEvent which is a POJO that contains information of access in/out of a building containing a card reader.

- The error codes and exceptions raised when accessing an invalid REST URL.

Mock JSON data has been passed to the GitListener and CalendarListener classes (into the listen functions).

Each of these components have dependencies on features such as the messaging queue. All of these features are mocked out to ensure each component is tested in isolation. These features were mocked through dependency injection which was provided by Spring.

7 Features that will not be tested

Features that have been provided by frameworks will not be tested. Operations such as adding an object to a messaging queue will not be tested. Mapping of JSON data to request objects as this is also done by frameworks.

8 Components to be mocked

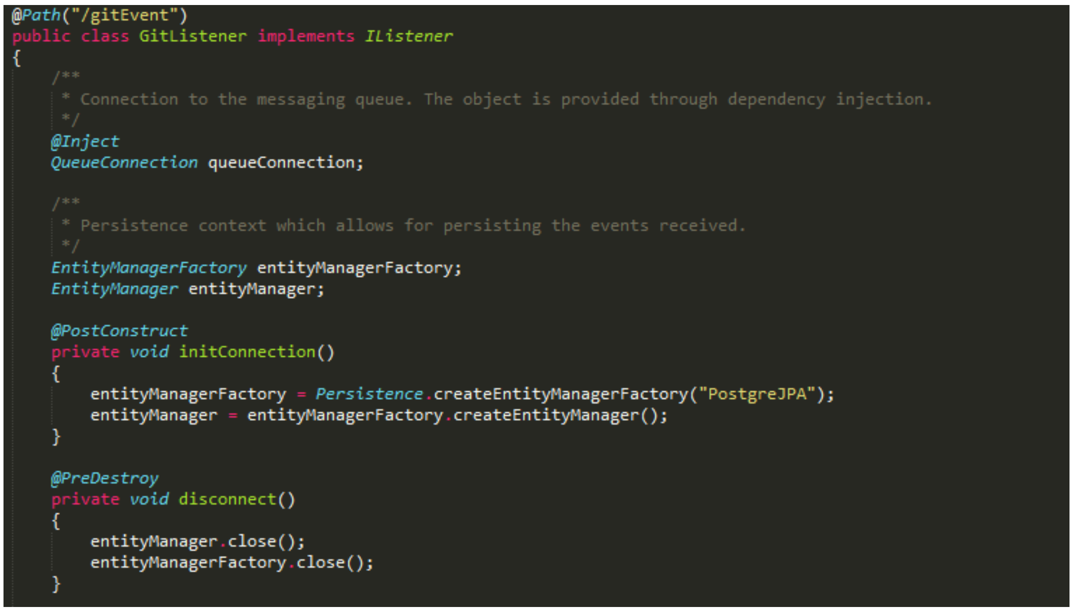
- The JPA entity manager was the first component that was mocked. When testing the possibility for persistence of the POJO's it would be best if the transactions do not affect the database. Thus every transaction that occurs with the mocked entity manager will leave the database system intact and unaffected. An alternate approach could have been to use an in-memory database - such as the one provided by H2 - however this was not deemed necessary.

- The second component to be mocked was the messaging queue provided by ActiveMQ. Once again we do not wish to have unnecessary objects in our queue that might affect actual program execution thus the default queue is replaced with a queue that does not retain objects.

- When testing the event listeners we can't wait for the event emitting systems to send out an event. So we send mock events to the listeners while testing, these mock events are structured in the same manner as there real-world counterparts.

9 Sample tests

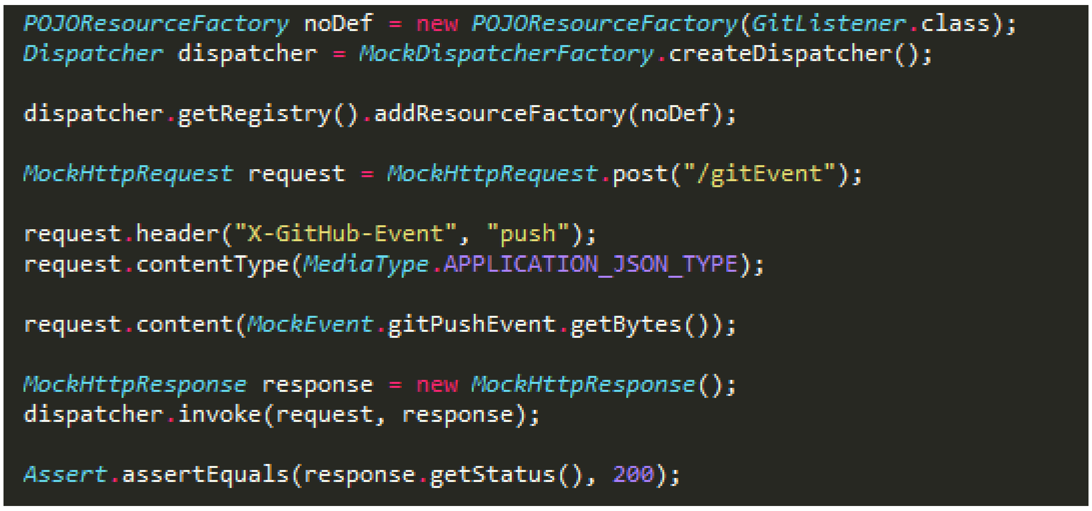
The following figure is the GitListener dependency injected queue which allows multiple events to come through and not be lost and the creation of the entity manager.



```java
@Path("/gitEvent")
public class GitListener implements IListener
{
    /**
     * Connection to the messaging queue. The object is provided through dependency injection.
     */
    @Inject
    QueueConnection queueConnection;

    /**
     * Persistence context which allows for persisting the events received.
     */
    EntityManagerFactory entityManagerFactory;
    EntityManager entityManager;

    @PostConstruct
    private void initConnection()
    {
        entityManagerFactory = Persistence.createEntityManagerFactory("PostgreJPA");
        entityManager = entityManagerFactory.createEntityManager();
    }

    @PreDestroy
    private void disconnect()
    {
        entityManager.close();
        entityManagerFactory.close();
    }
```

Figure 1: GitListener Dependency injection

The following figure is one of the GitListener Event tests.

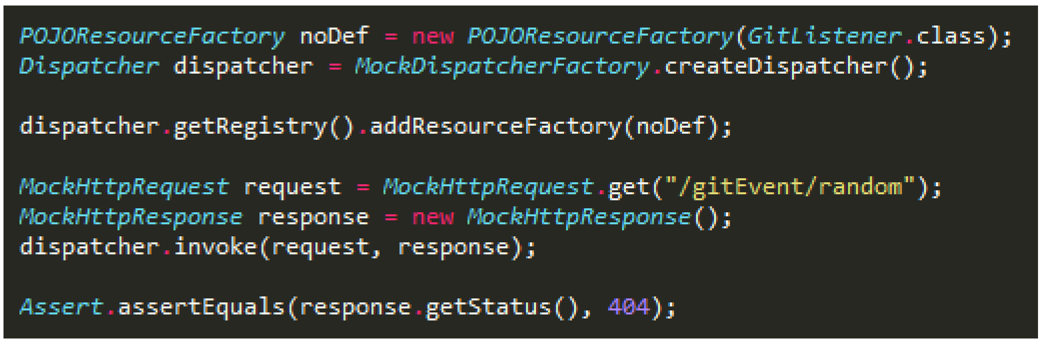


```java
POJOResourceFactory noDef = new POJOResourceFactory(GitListener.class);
Dispatcher dispatcher = MockDispatcherFactory.createDispatcher();

dispatcher.getRegistry().addResourceFactory(noDef);

MockHttpRequest request = MockHttpRequest.post("/gitEvent");

request.header("X-GitHub-Event", "push");
request.contentType(MediaType.APPLICATION_JSON_TYPE);

request.content(MockEvent.gitPushEvent.getBytes());

MockHttpResponse response = new MockHttpResponse();
dispatcher.invoke(request, response);

Assert.assertEquals(response.getStatus(), 200);
```

Figure 2: GitListener Push event test

In the above figure the *POJOResourceFactory* and *Dispatcher* are used to start up an embedded server which will allow for calls to be made to a particular URL, in this case \ gitEvent.

A post request is created and has the mock event as its content. This post request mirrors the post requests made by GitHub when sending events. Once the mock event data is loaded into the request, the request is sent. At the end, the response objects' HTTP status code is checked. This is checked in an assert statement, the value of the response should be 200 to indicate a successful retrieval.



```
POJOResourceFactory noDef = new POJOResourceFactory(GitListener.class);
Dispatcher dispatcher = MockDispatcherFactory.createDispatcher();

dispatcher.getRegistry().addResourceFactory(noDef);

MockHttpRequest request = MockHttpRequest.get("/gitEvent/random");
MockHttpResponse response = new MockHttpResponse();
dispatcher.invoke(request, response);

Assert.assertEquals(response.getStatus(), 404);
```

Figure 3: Exception handler for invalid URLs

10 Unit Test Report

10.1 Overview

All the test of the system has passed and the reason that a test may fail is if the database tables are not created properly as these tables are not created using JPA, since JPA destroys and creates the tables upon the command used for testing *mvncleantest*. We use Travis CI for our builds to ensure that all the build pass.

10.2 Test Cases

10.2.1 Test Class 1

`https://github.com/HyperPerform/hyper-perform-server/blob/develop/src/test/java/me/hyperperform/event/CalendarPersistenceTest.java`

CalendarPersistenceTest

- *jpaTest* - Test if the CalendarMeeting and CalendarProject POJOS persist.
- *QueryTest* - Test to see if the persisted objects were actually persisted properly and check the data integrity.

Result: **All Passed**

10.2.2 Test Class 2

`https://github.com/HyperPerform/hyper-perform-server/blob/develop/src/test/java/me/hyperperform/event/DatabasePopulatorTest.java`

DatabasePopulatorTest

- *databasePopTest* - This test is to insert mock data for all the tables into the database so that relevant reports can be generated.

Result: **All Passed**

10.2.3 Test Class 3

`https://github.com/HyperPerform/hyper-perform-server/blob/develop/src/test/java/me/hyperperform/event/PersistenceTest.java`

PersistenceTest

- *queryTest* - This creates a GitPush POJO and persists it thereafter query the database and check data integrity.
- *gitIssuePojoTest* - This creates a GitIssue POJO and persists it.
- *travisPojoTest* - This creates a TravisEvent POJO and persists it.

Result: **All Passed**

10.2.4 Test Class 4

Test case 1: Git Push Event Test
Objective: This test will ensure that the git listener REST endpoint is working and successfully receives events
Input: A mock git push event is send through to the link
Outcome: Once the event is sent through, a HTTP 200 response code is expected.

Test case 2: Git Issues Event Test
Objective: This test will ensure that the git listener REST endpoint is working and successfully accepts git issue events
Input: A mock git issue event is send through to the link

Outcome: Once the event is sent through, a HTTP 200 response code is expected.

Test case 3: Invalid link Test
Objective: This test is used to see if an appropriate response is sent for non-existing pages
Input: An intentional invalid link is accessed
Outcome: An HTTP 404 response code is expected.

Test case 4: Timezone Test
Objective: This test checks the adaptability of the Git Listeners in terms of timezones. Its aims to see if a correct timezone can be parsed from the received data.
Input: An alternative git push event is sent through
Outcome: An HTTP 200 response code is expected.

Test case 5: Travis Test
Objective: This test checks whether or not the travis listener is working. This is tested by sending through a mock event
Input: A travis event is sent through
Outcome: An HTTP 200 response code is expected.

Test case 6: Login Test
Objective: This test checks whether or not the login subsystem under the user management system is working.
Input: Mock user details are sent through to the REST endpoint
Outcome: An HTTP 200 response code is expected.

Test case 7: Access Test
Objective: This test checks whether or not the access listener is working. This is tested by sending through a mock event
Input: A mock access event is sent through
Outcome: An HTTP 200 response code is expected.

All these test cases above can be found on the following link: `https://github.com/HyperPerform/hyper-perform-server/blob/develop/src/test/java/me/hyperperform/rest/RestTest.java`

10.2.5 Test Class 5

All mock data can be found in the Appendix.
Test case 1: Create User Test
Objective: This test will create 2 different users with different roles and persist them into the database.
Input: There is no direct input into this test.
Outcome: The newly created users are successfully persisted to the database.

Test case 2: User Test
Objective: Check the integrity of the information of the above users and delete them from the database, to ensure no primary key violation.
Input: There is no direct input into this test.
Outcome: The newly created users data is as expected and then are successfully deleted from the database.

Test case 3: Registration Test
Objective: Check to see if a user can be added to the system with all valid information.
Input: A mock normalUser event is sent through
Outcome: An HTTP 200 response code is expected. There after the test will delete the user to ensure no primary key violation.

Test case 4: Invalid Name Test
Objective: Check to see if a user can be added to the system with no username.
Input: A mock noUsername event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: name”

Test case 5: Invalid Surname Test
Objective: Check to see if a user can be added to the system with no surname.
Input: A mock noSurname event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: surname”

Test case 6: Invalid Email Test
Objective: Check to see if a user can be added to the system with no email.
Input: A mock noEmail event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: email”

Test case 7: Invalid Role Test
Objective: Check to see if a user can be added to the system with an invalid role.
Input: A mock invalidRole event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: Role does not exist”

Test case 8: Invalid Position Test
Objective: Check to see if a user can be added to the system with an invalid position.
Input: A mock invalidPosition event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: Position does not exist”

Test case 9: Invalid GitUsername Test
Objective: Check to see if a user can be added to the system with an no GitUserName.
Input: A mock noGitUsername event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: gitUserName”

Test case 10: Invalid Password
Objective: Check to see if a user can be added to the system with an no password.
Input: A mock noPassword event is sent through.
Outcome: An HTTP 200 response code is expected with a response message ”Error: password”

All these test cases above can be found on the following link: `https://github.com/HyperPerform/hyper-perform-server/blob/develop/src/test/java/me/hyperperform/user/UserTest.java`

11 Conclusion

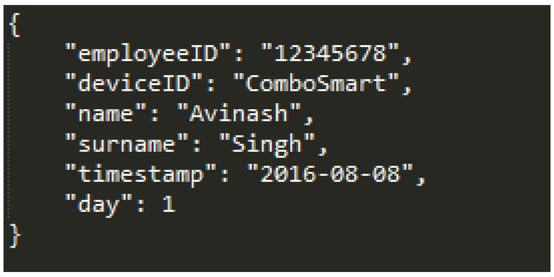
All of Test have passed as we made sure of it and since we using Travis CI which is an automated build system so if it fails we work on the issues and make the fixes. The limitations to these tests is that the require the relevant tables to already exist in the Postgres Database as JPA will not create these tables. A work around for this is to write a script or a unit test that will create these tables if they don’t exist before the rest of the unit tests are executed.

12 Appendix

12.1 Mock Events

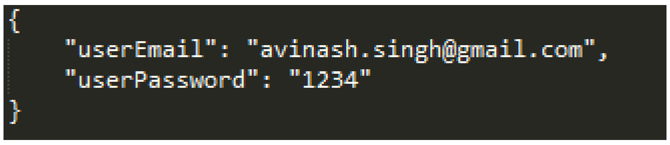
Please note that some of these mock events are very large. Due to this redundant data has been removed from the events so that they can be presented in this document.

The fully detailed mock events can be found through the following link: `https://github.com/HyperPerform/hyper-perform-server/blob/develop/src/test/java/me/hyperperform/event/MockEvent.java`



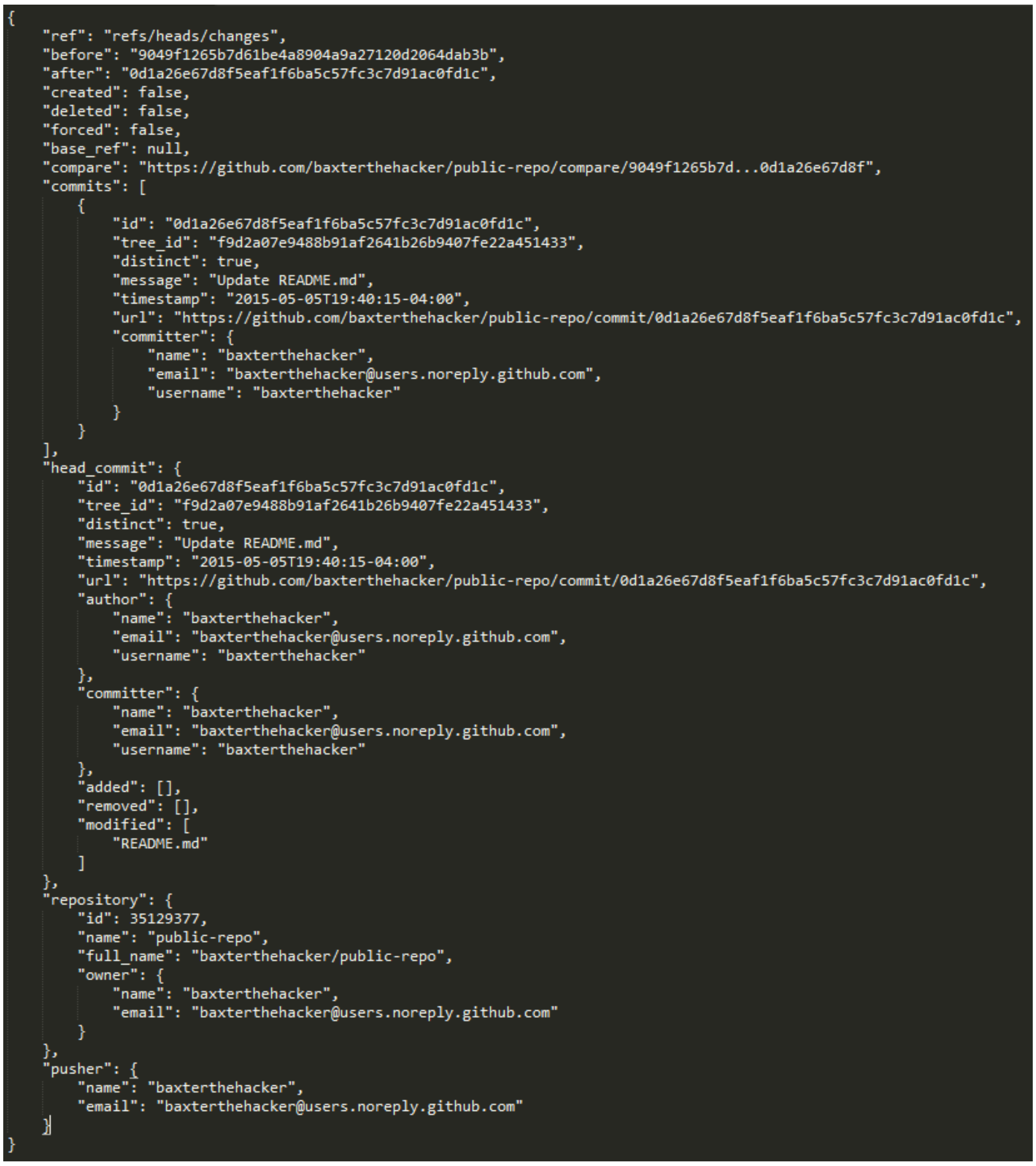
```
{
    "employeeID": "12345678",
    "deviceID": "ComboSmart",
    "name": "Avinash",
    "surname": "Singh",
    "timestamp": "2016-08-08",
    "day": 1
}
```

Figure 4: Access event payload



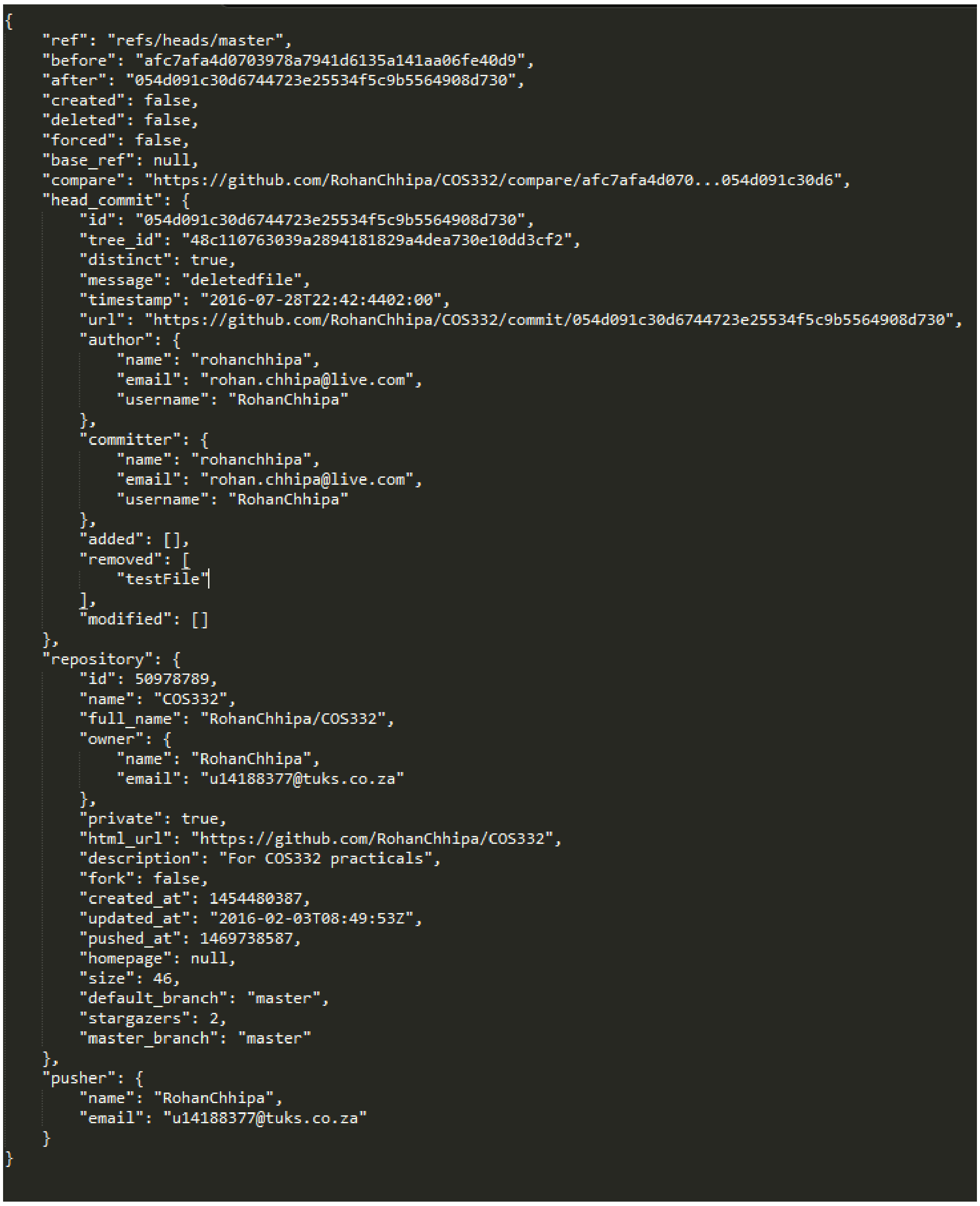
```
{
    "userEmail": "avinash.singh@gmail.com",
    "userPassword": "1234"
}
```

Figure 5: Login payload



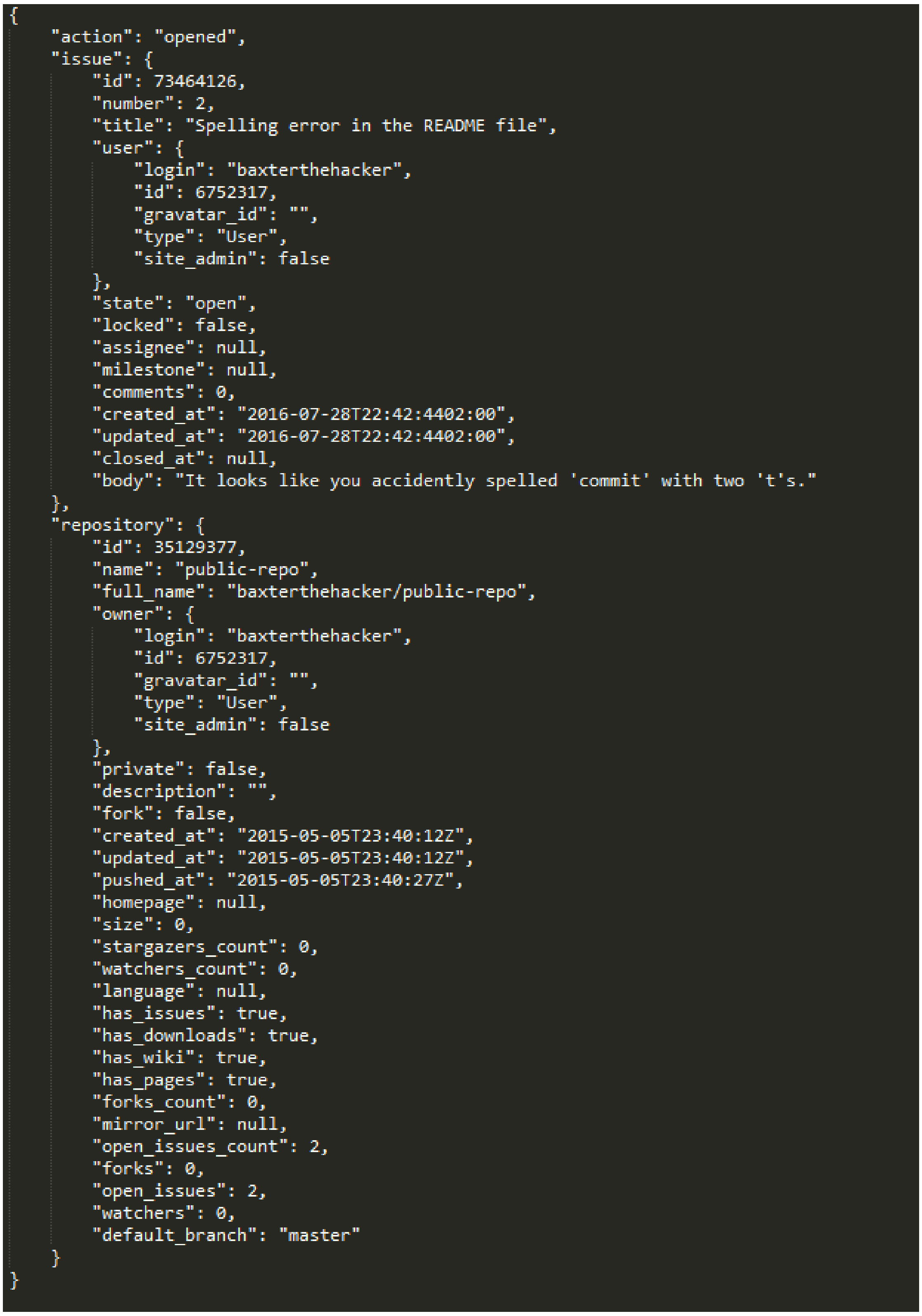
```
{
    "ref": "refs/heads/changes",
    "before": "9049f1265b7d61be4a8904a9a27120d2064dab3b",
    "after": "0d1a26e67d8f5eaf1f6ba5c57fc3c7d91ac0fd1c",
    "created": false,
    "deleted": false,
    "forced": false,
    "base_ref": null,
    "compare": "https://github.com/baxterthehacker/public-repo/compare/9049f1265b7d...0d1a26e67d8f",
    "commits": [
        {
            "id": "0d1a26e67d8f5eaf1f6ba5c57fc3c7d91ac0fd1c",
            "tree_id": "f9d2a07e9488b91af2641b26b9407fe22a451433",
            "distinct": true,
            "message": "Update README.md",
            "timestamp": "2015-05-05T19:40:15-04:00",
            "url": "https://github.com/baxterthehacker/public-repo/commit/0d1a26e67d8f5eaf1f6ba5c57fc3c7d91ac0fd1c",
            "committer": {
                "name": "baxterthehacker",
                "email": "baxterthehacker@users.noreply.github.com",
                "username": "baxterthehacker"
            }
        }
    ],
    "head_commit": {
        "id": "0d1a26e67d8f5eaf1f6ba5c57fc3c7d91ac0fd1c",
        "tree_id": "f9d2a07e9488b91af2641b26b9407fe22a451433",
        "distinct": true,
        "message": "Update README.md",
        "timestamp": "2015-05-05T19:40:15-04:00",
        "url": "https://github.com/baxterthehacker/public-repo/commit/0d1a26e67d8f5eaf1f6ba5c57fc3c7d91ac0fd1c",
        "author": {
            "name": "baxterthehacker",
            "email": "baxterthehacker@users.noreply.github.com",
            "username": "baxterthehacker"
        },
        "committer": {
            "name": "baxterthehacker",
            "email": "baxterthehacker@users.noreply.github.com",
            "username": "baxterthehacker"
        },
        "added": [],
        "removed": [],
        "modified": [
            "README.md"
        ]
    },
    "repository": {
        "id": 35129377,
        "name": "public-repo",
        "full_name": "baxterthehacker/public-repo",
        "owner": {
            "name": "baxterthehacker",
            "email": "baxterthehacker@users.noreply.github.com"
        }
    },
    "pusher": {
        "name": "baxterthehacker",
        "email": "baxterthehacker@users.noreply.github.com"
    }
}
```

Figure 6: Git push event payload



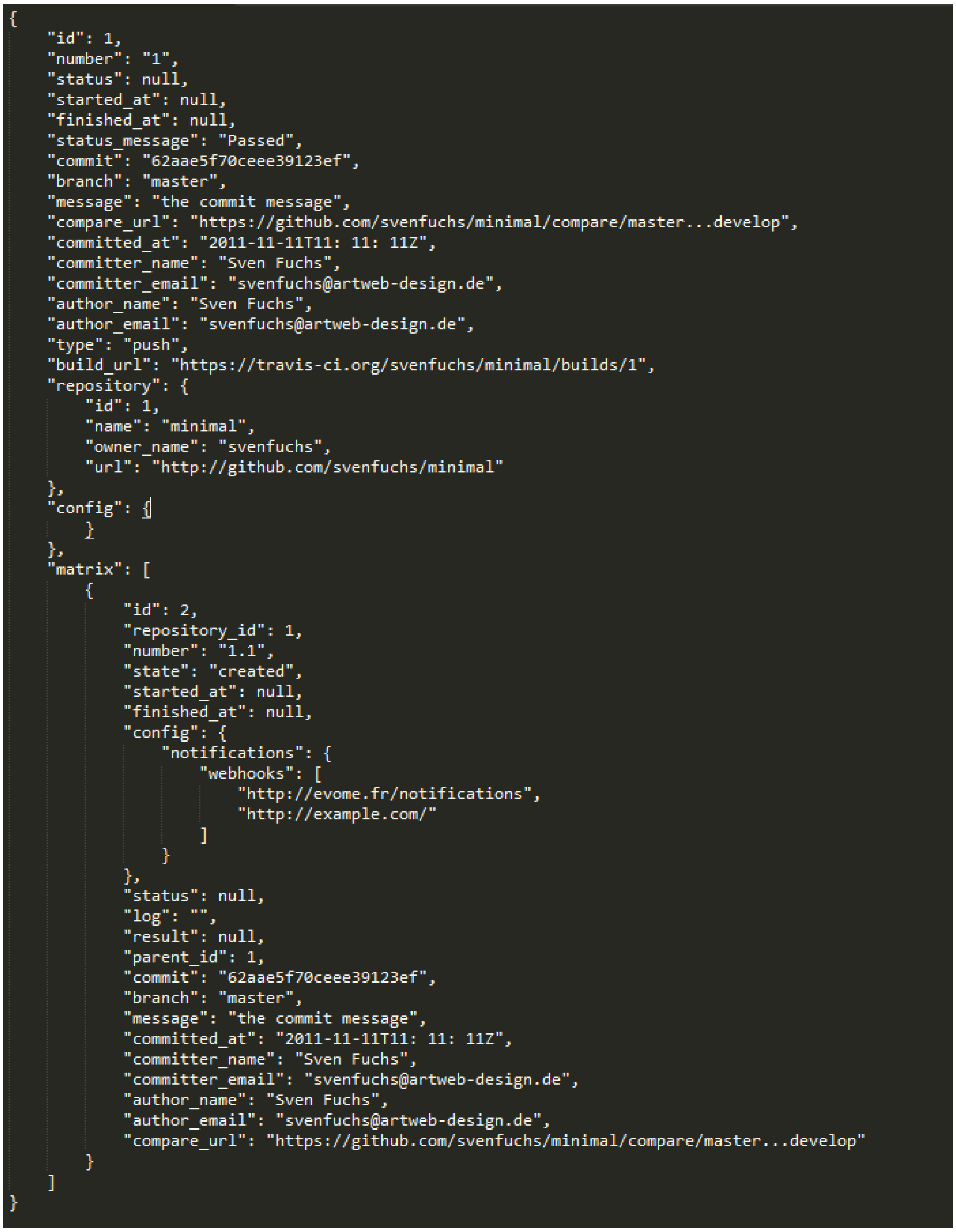
```
{
    "ref": "refs/heads/master",
    "before": "afc7afa4d0703978a7941d6135a141aa06fe40d9",
    "after": "054d091c30d6744723e25534f5c9b5564908d730",
    "created": false,
    "deleted": false,
    "forced": false,
    "base_ref": null,
    "compare": "https://github.com/RohanChhipa/COS332/compare/afc7afa4d070...054d091c30d6",
    "head_commit": {
        "id": "054d091c30d6744723e25534f5c9b5564908d730",
        "tree_id": "48c110763039a2894181829a4dea730e10dd3cf2",
        "distinct": true,
        "message": "deletedfile",
        "timestamp": "2016-07-28T22:42:4402:00",
        "url": "https://github.com/RohanChhipa/COS332/commit/054d091c30d6744723e25534f5c9b5564908d730",
        "author": {
            "name": "rohanchhipa",
            "email": "rohan.chhipa@live.com",
            "username": "RohanChhipa"
        },
        "committer": {
            "name": "rohanchhipa",
            "email": "rohan.chhipa@live.com",
            "username": "RohanChhipa"
        },
        "added": [],
        "removed": [
            "testFile"
        ],
        "modified": []
    },
    "repository": {
        "id": 50978789,
        "name": "COS332",
        "full_name": "RohanChhipa/COS332",
        "owner": {
            "name": "RohanChhipa",
            "email": "u14188377@tuks.co.za"
        },
        "private": true,
        "html_url": "https://github.com/RohanChhipa/COS332",
        "description": "For COS332 practicals",
        "fork": false,
        "created_at": 1454480387,
        "updated_at": "2016-02-03T08:49:53Z",
        "pushed_at": 1469738587,
        "homepage": null,
        "size": 46,
        "default_branch": "master",
        "stargazers": 2,
        "master_branch": "master"
    },
    "pusher": {
        "name": "RohanChhipa",
        "email": "u14188377@tuks.co.za"
    }
}
```

Figure 7: Alternative Git push event payload



```
{
    "action": "opened",
    "issue": {
        "id": 73464126,
        "number": 2,
        "title": "Spelling error in the README file",
        "user": {
            "login": "baxterthehacker",
            "id": 6752317,
            "gravatar_id": "",
            "type": "User",
            "site_admin": false
        },
        "state": "open",
        "locked": false,
        "assignee": null,
        "milestone": null,
        "comments": 0,
        "created_at": "2016-07-28T22:42:4402:00",
        "updated_at": "2016-07-28T22:42:4402:00",
        "closed_at": null,
        "body": "It looks like you accidently spelled 'commit' with two 't's."
    },
    "repository": {
        "id": 35129377,
        "name": "public-repo",
        "full_name": "baxterthehacker/public-repo",
        "owner": {
            "login": "baxterthehacker",
            "id": 6752317,
            "gravatar_id": "",
            "type": "User",
            "site_admin": false
        },
        "private": false,
        "description": "",
        "fork": false,
        "created_at": "2015-05-05T23:40:12Z",
        "updated_at": "2015-05-05T23:40:12Z",
        "pushed_at": "2015-05-05T23:40:27Z",
        "homepage": null,
        "size": 0,
        "stargazers_count": 0,
        "watchers_count": 0,
        "language": null,
        "has_issues": true,
        "has_downloads": true,
        "has_wiki": true,
        "has_pages": true,
        "forks_count": 0,
        "mirror_url": null,
        "open_issues_count": 2,
        "forks": 0,
        "open_issues": 2,
        "watchers": 0,
        "default_branch": "master"
    }
}
```

Figure 8: Git issues event payload



```
{
    "id": 1,
    "number": "1",
    "status": null,
    "started_at": null,
    "finished_at": null,
    "status_message": "Passed",
    "commit": "62aae5f70ceee39123ef",
    "branch": "master",
    "message": "the commit message",
    "compare_url": "https://github.com/svenfuchs/minimal/compare/master...develop",
    "committed_at": "2011-11-11T11: 11: 11Z",
    "committer_name": "Sven Fuchs",
    "committer_email": "svenfuchs@artweb-design.de",
    "author_name": "Sven Fuchs",
    "author_email": "svenfuchs@artweb-design.de",
    "type": "push",
    "build_url": "https://travis-ci.org/svenfuchs/minimal/builds/1",
    "repository": {
        "id": 1,
        "name": "minimal",
        "owner_name": "svenfuchs",
        "url": "http://github.com/svenfuchs/minimal"
    },
    "config": {
        }
    },
    "matrix": [
        {
            "id": 2,
            "repository_id": 1,
            "number": "1.1",
            "state": "created",
            "started_at": null,
            "finished_at": null,
            "config": {
                "notifications": {
                    "webhooks": [
                        "http://evome.fr/notifications",
                        "http://example.com/"
                    ]
                }
            },
            "status": null,
            "log": "",
            "result": null,
            "parent_id": 1,
            "commit": "62aae5f70ceee39123ef",
            "branch": "master",
            "message": "the commit message",
            "committed_at": "2011-11-11T11: 11: 11Z",
            "committer_name": "Sven Fuchs",
            "committer_email": "svenfuchs@artweb-design.de",
            "author_name": "Sven Fuchs",
            "author_email": "svenfuchs@artweb-design.de",
            "compare_url": "https://github.com/svenfuchs/minimal/compare/master...develop"
        }
    ]
}
```

Figure 9: Travis build event payload