Hyper Perform
Testing Document

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

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

Updated September 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 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.

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.

3 Execution of unit tests

The project was developed using Maven as a build tool. 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.

4 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.

5 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.

6 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.

7 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