Projekt: adit
Testplan

Oliver Dias odiaslal@hsr.ch, Fabian Hauser fhauser@hsr.ch, Murièle Trentini mtrentin@hsr.ch,
Nico Vinzens nvinzens@hsr.ch, Michael Wieland mwieland@hsr.ch

Änderungsgeschichte

Datum	Version	Änderung	Autor
23.02.2017	1.0	Erstellen des Dokuments	wie
15.03.2017	1.1	Hinzufügen Contracts	dia
19.03.2017	1.2	Contracts Entfernt und Review Dokument	hau, wie
27.03.2017	1.3	Exception Handling added	wie
03.05.2017	1.4	Anpassung Ordnerstruktur Testing Frontend	
30.05.2017	1.5	Dokument Review	wie

Inhalt

1. Einführung

1.1 Zweck

Der vorliegende Testplan beschreibt wie das System getestet werden soll. Es soll helfen, die Testbarkeit des Systems bereits vor der Entwicklungszeit zu betrachten, um eine optimale Architektur zu erzielen.

1.2 Gültigkeitsbereich

Der Gültigkeitsbereich beschränkt sich auf die Projektdauer des Moduls Engineering Projekt FS17. Das Dokument wird ausschliesslich HSR Intern verwendet.

1.3 Referenzen

Beschreibung	Name
Anforderungsspezifikation	2_Anforderungsspezifikation.pdf
Testprotokoll	\5_Testprotokolle\
Projektplan	1_Projektplan.pdf

2. Grundlegend

Der Entwicklungsprozess soll so organisiert sein, dass Tests fortlaufend geschrieben werden. Jeder Test soll systematisch, wiederholbar und automatisiert sein und genau einen Execution Path abdecken.

2.1 Testabdeckung

Wir legen grossen Wert darauf, dass relevante Codeabschnitte getestet werden. Tests für «funktionslosen Code» (z.B Getter und Setter) werden bewusst weggelassen, da man davon ausgehen kann, dass solch triviale Logik funktioniert. Des Weiteren werden Getter und Setter indirekt auch durch die restliche Funktionalität mitgetestet.

- Test anything that might break: Keine Tests für funktionslosen Code
- Test everything that does break: Test schreiben, die frühere Fehler abdecken.
- New Code is guilty until proven innocent

Die Testabdeckung wird auf dem Server mit dem Eclipse Plugin *EclEmma* sowie clientseitig mit *istanbul.js* überprüft. Wir streben nach einer Testabdeckung von 70%, wobei die Tests durch Qualität anstatt Quantität überzeugen sollen. Kann die gewünschte Abdeckung nicht eingehalten werden, gilt zu prüfen, ob die wichtige Funktionalität mit sauberen Äquivalenzklassen abgedeckt ist.

2.2 Integration

Durch die Integration des Test Frameworks in den Buildprozess ist garantiert, dass die Tests häufig ausgeführt werden und Fehler schnell erkannt werden. Durch das Verpacken in Docker-Container wird sichergestellt, dass sowohl überall die gleichen Abhängigkeiten und Umgebungen vorhanden sind.
Alle Tests werden mit Travis CI[1] sowie Docker automatisiert. In der Entwicklungsumgebung werden nur die Unit Tests ausgeführt, wohingegen auf dem Development Server zusätzlich die Integration Tests durchgeführt werden. Schlagen die Tests fehl, wird kein Deploy durchgeführt.

2.3 Mocking

Eigene Abhängigkeiten werden durch Mocks ersetzt. Als Mocking Framwork auf Serverseite wird Mockito[2] eingesetzt. Dabei gehen wir nach dem Grundsatz «Don't mock types you don't own». Für dieses Vorgehen gibt es zwei Gründe:

1. Wir wissen nicht, welche die korrekten Methodenaufrufe im Framework sind. Dies bedeutet, wir müssten z.B. den Hibernate Code analysieren und die verwendeten Methoden raten.
2. Wenn sich das Framework ändert, schlagen unsere Tests fehl, obschon die Funktionalität weiterhin in Ordnung wäre.

Der Data Access Layer wird deshalb nur mittels Integration Tests und einer Testdatenbank getestet. Auf Clientseite werden Mocks/Stubs direkt mit Angular TestBed geschrieben.

[1] https://travis-ci.org/
[2] http://site.mockito.org/

2.4 Definition of Done

Die Defintion of Done definiert, wann ein Stück Software eingecheckt werden darf:

- Keine Warnings
- Alles Unfertige ist markiert mit «TODO»
- Es existieren sinnvolle Units Tests, die erfolgreich laufen
- Kein auskommentierter Code
- Schwierig zu verstehende Codeabschnitte sind mit Kommentaren versehen
- Alle qualitätssichernden Plugins akzeptieren den Code
- Codereview wurde durchgeführt
- Das dazugehörende Issue ist in Redmine auf «Closed» gesetzt.

2.5 Exception Handling

Exceptions werden serverseitig gefangen und innerhalb des Backends an den globalen Exception Handler von Spark weitergereicht. Jede applikations-interne Exception wird mit einem ErrorCode getaggt, und beim globalen Exception Handler auf einen passenden HTTP Response Code gemappt. Das Frontend zeigt dem Benutzer dann eine aussagekräftige Meldung. Die Meldung soll, falls möglich, Hilfestellungen für das weitere Vorgehen enthalten.

Clientseitig werden Exception, sofern benutzerrelevant, direkt angezeigt. Dafür existiert ein Template, das bei einem Fehler gerendert wird.

3. Unit und Integration Tests

Java-Code soll mittels JUnit 4.0 automatisch getestet werden. TypeScriptCode wird mittels Jasmine getestet. Alle Tests werden durch Travis CI automatisch ausgeführt. Feature Branches werden nur übernommen, wenn die Tests erfolgreich durchlaufen.

3.1 Teststruktur

Unit Tests werden nach dem Prinzip Arrange, Act, Assert, Teardown geschrieben. Dies erleichtert das Lesen der Tests.

3.2 Projektstruktur

Java:
Apache Maven gibt die Projektstruktur für die Tests relativ stark vor. Unit sowie Integration Tests werden im Ordner /src/test abgelegt. Die Testklassen befinden sich immer im selben Package wie die zu testenden Klasse.

Typescript:
Die Ordnerstruktur für Jasime Tests wiederspiegelt die Struktur des zu testenden TypeScript Codes. Jedoch wird übersichtshalber nur die oberste Ordnerstruktur übernommen. Componenten-Tests bekommen keinen eigenen Unterordner im /test/ Ordner.

Source Code	Test Code
src/	tests/
src/app/ **./components** **./utils** **./services** **./widgets**	tests/app/ ./components ./utils ./services ./widgets

3.3 Automatisierung

3.3.1 Maven Integration

Unit sowie Integration Tests werden automatisch durch Apache Maven[3] ausgeführt.
Surefile Plugin: Führt die JUnit Tests automatisch aus
Filesafe Plugin: Führt die JUnit Integrations Tests automatisch aus.

3.3.2 Webpack Integration

Jasmine Test Code wird während des Build-Prozesses von Webpack[4] automatisch via Karma[5] Test Runner ausgeführt.

[3] https://maven.apache.org/
[4] https://webpack.github.io/
[5] https://karma-runner.github.io/1.0/index.html

4. Systemtests

Systemtests werden anhand der beschriebenen Use Cases aus der Anforderungsspezifikation[6] durchgeführt. Das Testprotokoll[7] ist als Excel Tabelle im Projektverzeichnis abgelegt. Systemtests werden von allen Teammitglieder zum ersten Mal nach Milestone MS4, sowie abschliessend bis MS8 durchgeführt.

5. Nichtfunktionale Tests

5.1 Usability Tests

Usability Tests werden vom Server Team durchgeführt, da diese nicht direkt bei der Entwicklung des Frontend beteiligt sind. Die Resultate werden im entsprechenden Testprotokoll[8] protokolliert. Usability Test sind in einem einwöchigen Sprint bis MS4 eingeplant.

5.2 Performance Tests

Performance Tests werden manuell mit dem HTTP Load Testing Tool Vegeta[9] durchgeführt. Performance Tests sollen als Teil des finalen Systemtests durchgeführt werden.

5.3 Coding Conventions

Die Qualität des Source Code wird mit unterschiedlichen Tools sichergestellt:

Java: Checkstyle Plugin[10]
Typescript: TSLint[11]
SCSS: sass Lint[12]

5.4 Statische Code Analyse

Für die statische Code Analyse wird serverseitig das Plugin FindBugs[13] in Maven integriert. Auf Clientseite führt TSLint eine statische Analyse aus.

5.5 Metriken

Für die Visualisierung der Code Metriken setzen wir auf SonarQube, welche mit diversen Plugins erweitert wird.

FindBugs : https://docs.sonarqube.org/display/SONARQUBE45/FindBugs+Plugin
SonarTsPlugin: https://github.com/Pablissimo/SonarTsPlugin

---

6 ..\..\Anforderungen\Anforderungsspezifikation.docx
7 Testprotokoll.xlsx
8 Testprotokoll.xlsx
9 https://github.com/tsenart/vegeta
10 http://checkstyle.sourceforge.net/
11 https://palantir.github.io/tslint/
12 https://www.npmjs.com/package/sass-lint
13 http://findbugs.sourceforge.net/

6. Abnahmetests

Es werden keine Abnahmetests durchgeführt, da für dieses Projekt kein realer Kunde existiert.