ECSE Software Validation Term Project

Part A Exploratory Testing of Rest API

Application Under Test

The application under test is a "rest api todo list manager" which may be run as a local host.

A copy of the application is available in myCourses content under topic project.

The application is made available by Alan Richardson and can be found online at:

https://github.com/eviltester/thingifier/releases

Launch the rest api todo list manager with the command:

```
java -jar runTodoManagerRestAPI-1.5.5.jar
```

Basic documentation about the api todo list manager is found at:

http://localhost:4567/docs

A swagger description of the rest api todo list manager is referenced in the documentation and can be found at the link:

http://localhost:4567/docs/swagger

Individuals interested in learning about or using Swagger may benefit from a free individual account at:

https://swagger.io/tools/

Exploratory Testing

Team members are required to use Charter Driven Session Based Exploratory Testing to study the behavior of the "rest api todo list manager".

Team members may do exploratory testing sessions individually or in pairs. All team members must participate in exploratory testing sessions.

Exploratory testing sessions should be timeboxed at 45 minutes.

The charter is:

```
Identify capabilities and areas of potential instability of the "rest
api todo list manager".

Identify documented and undocumented "rest api todo list manager"
capabilities.

For each capability create a script or small program to demonstrate
the capability.

Exercise each capability identified with data typical to the intended
use of the application.
```

- If teams have one member, they should focus on capabilities related to *todos*.
- If teams have two members, they should focus on capabilities related to *todos* and *projects*.
- If teams have three members, they should focus on capabilities related to *todos* and *projects* and *categories*.
- If team members have four members, they should focus on capabilities related to *todos* and *projects* and *categories* and the *interoperability* of these capabilities.

Deliverables from each session should include:

- Session notes
  - Explicitly reference any scripts, programs, screen shots, video clips, spreadsheets or any other files used in or created during the session.
  - Include name and student id number and email address of session participants.
- Include any files created in the session.
- Summary of session findings
  - This is a bullet list created when the session findings are reviewed.
  - This list answers the question what we learned.
- List of concerns identified in session.
- List of new testing ideas identified in session.

Unit Test Suite

Team members are required to implement a suite of unit tests using an open-source unit test tool such a Junit. The unit test tool selected depends on which programming language the team chooses to create the unit tests. All unit tests from a team must be in the same suite using the same programming language and the same unit testing tool.

Teams must not use automatically generated unit testing code.

At least one separate unit test module is required for each API identified in the exploratory testing. This must include at least one unit test module for each documented API and at least one unit test module for each undocumented API discovered during exploratory testing.

Confirm the API does what is it supposed to do.

Identify bugs in the API implementation if the actual behavior is different from the documented behavior.

Note for cases when the API behavior is different from the documentation please include two separate modules one showing the expected behavior failing and one showing the actual behavior working. *The API may not behave as documented, but it may still allow the operation to succeed in an undocumented manner.*

Confirm the API does not have unexpected side effects. *It is sufficient to show that changes to data in the system are restricted to those which should change based on the API operation*.

Confirm that each API can generate payloads in JSON or XML.

Confirm that command line queries function correctly.

Confirm return codes are correctly generated.

Unit test modules must:

- Ensure the system is ready to be tested
- Save the system state
- Set up the initial conditions for the test
- Execute the tests
- Assess correctness
- Restore the system to the initial state
- Run in any order
- Use clean, well-structured code follow guidelines of Bob Martin on Clean Code.

Additional Unit Test Considerations

Ensure unit tests fail if service is not running.

Include at least one test to see what happens if a JSON payload is malformed.

Include at least one test to see what happens if an XML payload is malformed.

For each API *identified in the exploratory testing* include tests of invalid operations, for example, attempting to delete an object which has already been deleted.

Additional Bug Summary Considerations

The project team should define a form used to collect bug information which includes at least the following elements.

- Executive summary of bug in 80 characters or less
- Description of bug
- Potential impact of bug on operation of system
- Steps to reproduce the bug

Teams can choose any tool to track bug information.

Unit Test Suite Video

Show video of all unit tests running in the teams selected development environment.

Include demonstration of tests run in random order determined using pseudo random number generation to dictate the order.

Written Report

Target report size is between 5 and 10 pages.

Summarizes deliverables.

Describes findings of exploratory testing.

Describes structure of unit test suite.

Describes source code repository.

Describe findings of unit test suite execution.

Part B Story Testing of Rest API

Define 5 user stories per team member related to using the API explored in Part A of the project. Use the style:

> As a user I want to complete some operation to get some value.

Your team may choose any domain.

Story Test Suite

At least three acceptance tests must be defined for each story. At least one of each of the following types of acceptance tests are required.

- Normal flow
- Alternate flow
- Error flow

Story tests are defined as scenario outline gherkin scripts, in feature files. Each scenario outline gherkin script can be executed many times with different values for variable data.

Story tests should include a background section to set the initial conditions for the gherkin script.

Story Test Automations

Project teams will use the open-source story test automation tool cucumber, or any similar tool which can parse and execute gherkin scripts.

Team shall implement step definitions which control the to do list manager rest api studies in Part A of the project.

Step definitions should be built as a library, elements of which can be reused in several different acceptance tests.

Code in step definitions should:

- Use clean, well-structured code follow guidelines of Bob Martin on Clean Code.

Each gherkin script should

- Assess correctness.
- Restore the system to the initial state upon completion.
- Run in any order.

Additional Story Test Considerations

Ensure story tests fail if service is not running.

Additional Bug Summary Considerations

The project team should use the same format as in part A to report any bugs identified. If new bugs are found, please indicate which story they are related to.

Story Test Suite Video

Show video of all story tests running in the teams selected development environment.

Include demonstration of tests run in different orders.

Written Report

Target report size is between 5 and 10 pages.

Summarizes deliverables.

Describes structure of story test suite.

Describes source code repository.

Describe findings of story test suite execution.

Part C Non-Functional Testing of Rest API

Two types of non-functional testing will be implemented in part C of the project. Performance Testing using Dynamic Analysis and Static Analysis of the source code.

Performance Testing

Create a program which creates objects based on the API explored in part A of the project. Populate these objects with random data.

Adapt unit test code from Part A to measure the time to complete a create, delete or change object operation.

Perform the following experiments:

- As the number of objects increases measure the time required to add, delete, or change the objects.

Use operating system tools or open-source tools to track CPU percent use and Available Free Memory while each experiment takes place.

Operating system tools that may support dynamic analysis include:

```
Windows perfmon

Mac Activity Monitor

Linux vmstat
```

Static Analysis

The source code of the Rest API to do list manager will be studied with the community edition of the Sonar Cube static analysis tool.

```
https://www.sonarqube.org/downloads/
```

Recommendations should be made regarding changes to the source code to minimize risk during future modifications, enhancements, or bug fixes. Teams should look at code complexity, code statement counts, and any technical risks, technical debt or code smells highlighted by the static analysis.

Performance Test Suite Video

Show video of all performance tests running in the teams selected development environment.

Written Report

Target report size is between 5 and 10 pages.

Summarizes deliverables.

Describes implementation of performance test suite.

> Include charts (excel or other tool) showing transaction time, memory use and cpu use versus number of objects for each experiment.
>
> Summarize any recommendations for code enhancements related to results of performance testing.

Highlight any observed performance risks.

Describes implementation of static analysis with Sonar Cube community edition.

Include recommendations for improving code based on the result of the static analysis.

Summary of clean code guidelines from Bob Martin.

Understandability tips

- Be consistent. If you do something a certain way, do all similar things in the same way.
- Use explanatory variables.
- Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.
- Prefer dedicated value objects to primitive type.
- Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
- Avoid negative conditionals.

Names rules

- Choose descriptive and unambiguous names.
- Make meaningful distinction.
- Use pronounceable names.
- Use searchable names.
- Replace magic numbers with named constants.
- Avoid encodings. Do not append prefixes or type information.

Functions rules

- Small.
- Do one thing.
- Use descriptive names.
- Prefer fewer arguments.
- Have no side effects.
- Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

Comments rules

- Always try to explain yourself in code.
- Don't be redundant.
- Don't add obvious noise.
- Don't use closing brace comments.
- Don't comment out code. Just remove.
- Use as explanation of intent.
- Use as clarification of code.
- Use as warning of consequences.

Source code structure

- Separate concepts vertically.
- Related code should appear vertically dense.
- Declare variables close to their usage.

- Dependent functions should be close.
- Similar functions should be close.
- Place functions in the downward direction.
- Keep lines short.
- Don't use horizontal alignment.
- Use white space to associate related things and disassociate weakly related.
- Don't break indentation.
- Objects and data structures
- Hide internal structure.
- Prefer data structures.
- Avoid hybrids structures (half object and half data).
- Should be small.
- Do one thing.
- Small number of instance variables.
- Base class should know nothing about their derivatives.
- Better to have many functions than to pass some code into a function to select a behavior.
- Prefer non-static methods to static methods.

Tests

- Readable.
- Fast.
- Independent.
- Repeatable.

Avoid Code smells

- Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
- Fragility. The software breaks in many places due to a single change.
- Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
- Needless Complexity.
- Needless Repetition.
- Opacity. The code is hard to understand.