Spencer Plant

Find an Expert Sprint 3 Eval

21st March 2018

Reviews

User Stories

- US 2.01: “and not have that my information overwritten” -- I think this is grammatically incorrect. You may want to reword this to avoid having things misconceived.
- US 2.03: “the forum” isn’t very specific. If you specify which forum the user story is talking about, that can clarify things.

Overall, good work. It looks as though you took my comments from the previous sprint’s evaluation fixed some things.

Storyboards

Storyboards have been updated to reflect the changes to the User Stories. I noticed there aren’t any storyboards for the “nice to have” user stories; they were probably omitted on purpose, but it would still be nice to see them in case these stories come to fruition. If you do make up these storyboards, you can keep them separate from the “need to have” user stories for organizational reasons.

Architecture Design

The architecture for the project has been updated. Some things I noticed:

- In the Scraper class diagram, the get_content() method in ProfileScraper should be renamed to get_scrapps() to be consistent with the code and related classes
- Scrapp.py is inconsistent with the UML (has extra variables *data_source* and *formatted_name*, missing variable *keywords,* has extra setter methods)
- Scraper.py has one more method *get_scrapps* than what the UML suggests
- builder.py and parser.py import Stack using different ‘from’ clauses: (“bfex.components.search_engine.stack” vs “.stack”). This shouldn’t be an issue for running things, but consistency is good

- KeywordGenerator UML doesn't specify the variable *allowed_ids*, but that's in the code
- Variable names defined in Publication, Faculty, and Grant UML don't line up one-to-one with the code in *models.py*. Grant UML is very different than what is seen in the Grant model
- Workflow UML mentions variable *previous_result* but the code uses the name *last_result*
- I noticed that you have updated the main architecture design image to specify you are using React (was previously "*Django?*")

Something I forgot to check for during the last sprint evaluation was your indentation style. I know the tabs vs. spaces debate is divisive, but what's important at the end of the day is that you consistently use your choice for everything. Obviously, Python won't even run if indentation is mixed, but I still checked to see if every file used spaces for indenting (your chosen indenting method). They did, so good work.

Tests

Feedback from Sprint 2 Evaluation was addressed. More tests were made, and they all look great.

Test Plan

I'll lay out a test plan for the team, based off the IEEE 829-2008 Standard.

Overview

This plan is for the team to follow to their discretion to better organize, schedule, and use appropriate testing methods.

Scope

What to test: Your current collection of tests covers most of the important areas I would suggest to test. For the sake of documentation, I'll specify here (points in *italics* have not yet been implemented).

- Test all different key generation approaches
- Test all different scraping approaches and associated classes and methods
- Test persisting data
- Test your custom utility functions

- *Test User Interface*

For testing user interface, I understand why you haven't done this yet. From what I can tell, UI is still not stressed as very important, and you only have two very small template files currently anyway. However moving forward with more UI elements, you'll want to include UI testing. There are tools out there to make this easier through various means, like by allowing you to write tests in just-about plain english, and by taking screenshots at every step of the testing process. Symfony is an example of a tool you can use for UI testing.

What level to test: You have a pretty good collection of low-level unit tests, so it might be good to include one or two higher level tests to test how these units interact with one another.

Approaches

Black box manual system testing: You probably already do this yourselves, but it's good to also bring in outside people to test things. They won't be as close to the code and won't be tempted to subconsciously choose "happy paths" when testing.

Automated unit tests: You have these. Good work!

Performance tests: These will be able to let you know if say a certain key generation approach or search is slower than others. This can help in the long run by pointing out potential issues related to speed before truly launching.

Pass/Fail Criteria

This can be just about whatever you so choose, but if you document your test pass/fail criteria, it's much easier to stay strict to it and keep a consistent quality of code.

Resourcing

Staff: I don't think you need to train or hire any new team members at the current stage in the project, so disregard this.

Time: I suggest devoting a specific hour or two weekly (or some other interval that fits) entirely for testing.

Test Environment

You'll be testing the software rather than the hardware it runs on, so the test environment can be just about any server hosting your app. You'll probably want to test both locally and deployed versions of the app.

Network dependencies: since the information the app access is publicly available, you shouldn't have to worry about any dependency on a specific network. Naturally, internet access is required though.

Tools: If you do work on the UI and implement UI testing, I recommend using a tool like Symfony for that.

Risks

Since this project doesn't include anything potentially life threatening (software controlling an irradiation machine in a hospital would be something potentially life threatening, for example), there aren't any apparent risks associated.

Tester Independence

When doing blackbox testing, consider having outside parties test the application. Programming-minded friends and colleagues can be useful in trying to test the application for potential break-points.

Rationale

Provide reasoning behind each test you perform and document this so that identical tests aren't carried out, which would waste time.

Related Artifacts/Links

If you set up a documentation method like JavaDoc, you can link to appropriate documentation (i.e. relevant UML). You can also link to relevant GitHub issues in your issue tracker.

Bug Reports

- Include steps to reproduce the bug
- Specify what platform the bug occurred on
- Specify what the expected output should be
- Specify what the actual output was
- Include error code/stack trace
- Try to avoid submitting multiple bug reports for the same issue
- Try not to guess the cause of the bug within the bug report
- Include screenshot if applicable (i.e. UI bugs)

Coverage

You have already implemented a form of code coverage, which is great! As I understand it, it is a version of coverage where code is considered “covered” if the application gets to and runs that line during a standard run.

Test coverage is a little different. For this project, I will want to see if the existing unit tests you have written completely cover their respective components within the application. For the next sprint, I will deliver my findings.

General

Good work team! I’m super impressed, as always.