SeedIt Test Plan

Table of Contents

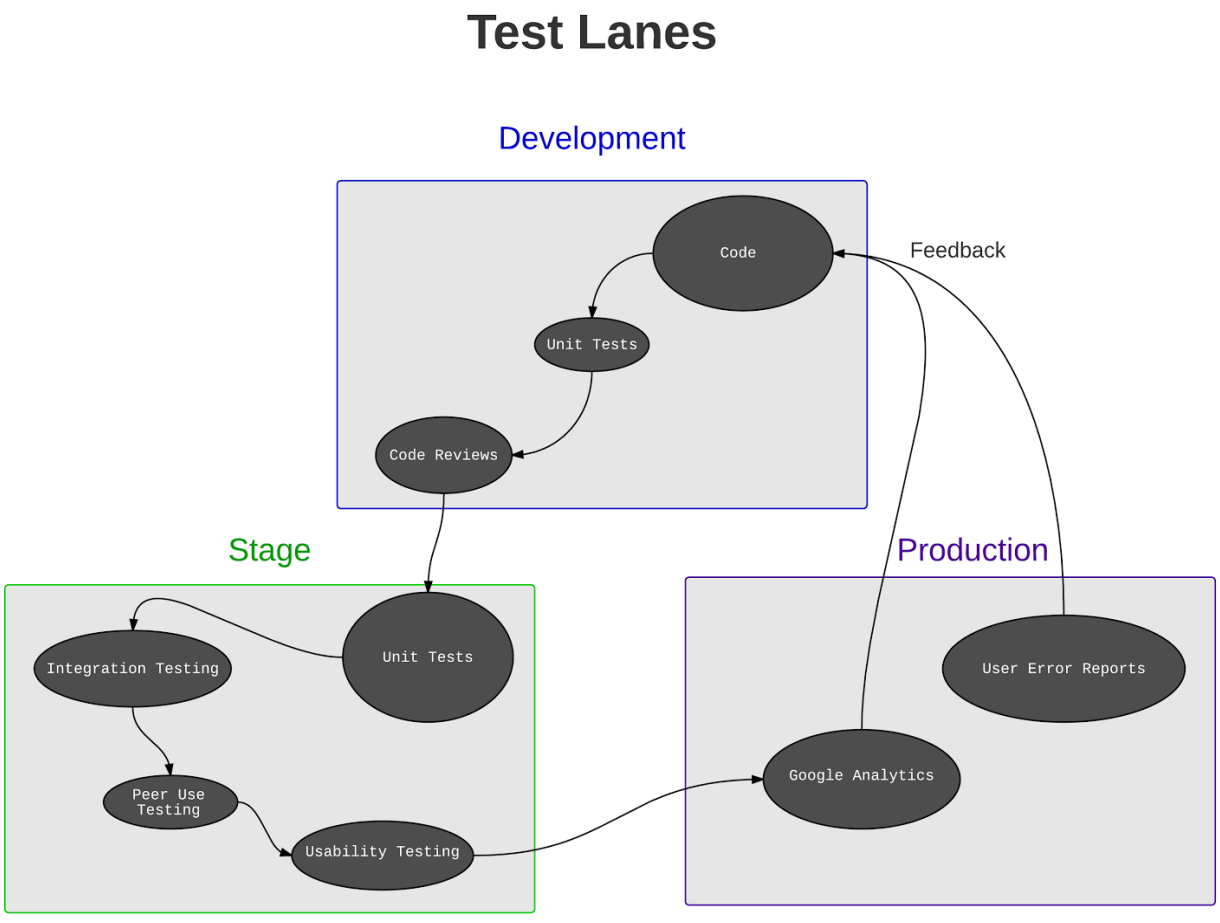
Introduction

According to our college Professor, Dr. Knutson, the amount of process that should be used in a project is just enough to optimize productivity. More process will clog the productivity of a small team while less process will inevitably produce poor code. SeedIt's test plan had to ultimately balance these two extremes in order to keep SeedIt's developers productive with quality code. SeedIt's test plan primarily focuses on finding and documenting defects at all stages of development rather than during a pre-defined testing phase.

The phases of development for SeedIt will occur in an iterative fashion: the development phase, staging phase, and production phase. These phases will be repeated perpetually with each iteration existing to keep SeedIt's development team responsive to user feedback and changes in the marketplace. These three phases will be represented as three separate git branches:

1. Code will be pushed on the **development branch** once it is written, code reviewed by other developers, and has passed basic unit tests.
2. Code will be pushed onto the **staging branch** once it is ready to be tested more rigorously by the dedicated testing staff, who will perform unit tests, automation tests, and integration tests.
3. Code will then be pushed onto the **production branch** once it has satisfactorily passed the dedicated testing staff's tests.

The important features we will rigorously test as well as a description of the process and technologies used to find defects in these three phases is explained in more detail in the document below.



Feature Correctness

In order to deliver a product that provides an enjoyable, consistent, and appreciable experience, SeedIt testing will focus on ensuring the delivery of features that embody this goal. All features will have a "standard of correctness" against which they will be tested. In the following list, we give examples of the standard of correctness for some of the features critical to the success of SeedIt. We also give examples of how these standards of correctness will be tested using the testing technologies we have selected. We also mention in what stage of our iterative development process those technologies will be used. Those technologies mentioned in this section are explained in more detail in the Frontend UI testing, Frontend Integration testing, Backend Testing and UX testing sections below this one. These are the standards of correctness and the technologies that will be applied to help us meet them:

- Trending Ideas (Feature 11)
    - Trending ideas will be reflective of the most popular ideas on SeedIt within the last day, and are defined as the ideas with the most up-votes within the last day of the time of the request.
    - The metric of "one day" should be responsive and as SeedIt grows in popularity should be adjusted downward, so that users are truly seeing the most popular ideas.
    - ❖ We will use Karma in the stage lane to make sure the correct trending ideas are retrieved from the server. We will also use Karma to make sure that the trending ideas are sorted correctly to the user. We will do this by verifying, for each search option, that the list is sorted correctly according to the user option selected.
    - ❖ We will use Protractor in the development and stage lanes to make sure the trending ideas in the main page are correctly displayed according to various user inputs in the search bars, tag list, and "sort by" drop-down box. This will be done (with Protractor) by programmatically clicking tags that appear on the page, typing in the search bar, and choosing a "sort by" option in different orders and then verifying the trending ideas displayed fit those options correctly.
    - ❖ We will use Google Analytics in the production lane to determine whether the trending tags have generated enough interest in our user-base. This will be done by measuring how many tags the average user clicked on during a visit to the site. If the users infrequently click on tags, this may be an indication that the tags are not displayed in an attention-grabbing manner.
    - ❖ [**TODO**::ADD HOW WE PLAN ON ENSURING CORRECTNESS]

- Tags (Feature 15)
    - Every seed can have one or more tags.
    - Tags serve as a means for categorizing ideas and enabling an effective user browsing experience.
    - Tags can be added to ideas
    - ❖ We will use selenium in the stage lane to make sure that we can find a seed based on a tag associated with it. This will be done by programmatically typing a tag name in a search bar and validating that the seed with the tag name appears in the results.

- ❖ We will use Protractor in the development and stage lanes to make sure that clicking on a tag will produce the correct branches inside of the main branch view. We will do this by running a protractor test in which a “tag” is clicked on, then reviewing the branches listed in the main view to ensure that they all have the tag which was clicked.
- ❖ [**TODO**::ADD HOW WE PLAN ON ENSURING CORRECTNESS]

- User Accounts (Feature 13)
  - Users will be able to securely login to SeedIt with a username and password in order to post ideas, up-vote other ideas, etc.
  - The information a user provides will be secured by his/her credentials and he/she will have the ability to update their account information as he/she sees fit.
  - ❖ Using the Mocha framework on the database, we’ll be able to drive unit tests that will ensure the correctness of the user accounts feature in the stage lane.
  - ❖ We will use Protractor in the development and stage lanes to test whether the proper elements appear in the page when the user is logged in, such as a “logout button.” This will be done by programmatically logging into the website and checking that certain UI elements appear in the page (“view profile”,”logout”) while others (such as “log in”) do not.
  - ❖ We will use Karma in the stage lane to determine that the login/logout functionality successfully logs in the user to the website. This will be done by creating a test user account and determining that the test user can successfully submit new branches to the server. We will also ensure that the user cannot edit branches that he/she is not authorized to edit.

- Trending Tags (Feature 2)
  - Similar to trending ideas, trending tags will represent the tags of the ideas that have received the most up-votes ideas within the last day. This will connect users to trending categories of ideas.
  - ❖ The Mocha framework will also ensure that the Trending Tags feature will be correct. Several unit tests will be utilized during the stage lane.
  - ❖ We will use Karma in the stage lane to make sure that our tags are correctly sorted according to popularity as the client views them. We will do this by programmatically validating that the JSON object containing the tags is in correct order.

- Search (Feature 12)
  - A database of tags and titles will drive the user’s ability to search SeedIt. When utilizing SeedIt search, a user search be “auto-filled” with SeedIt matches for their search.
  - For example, “ba” would yield an auto suggest list of “basketball”, “baseball”, “bass”, “bank”, etc. representing all tags or ideas that are matches for “ba”.
  - Additionally, SeedIt search (and all other user input) will be resistant to script injection, responsive to varying character sets, and responsive.
  - ❖ We will use Selenium in the stage environment to run various macro tests on the search bar in the browser to ensure the correctness of this functionality.

- ❖ We will use Protractor in the development and stage lanes to determine that the correct branches are placed in the main branch view whenever a user types into the search bar. We will do this by using Protractor to programmatically type characters into the search bar and verifying that the results which appear match the item being searched. For example, “fi” should match “filling” and “fishing,” but once the user types “fish”, Protractor will verify that “filling” no longer appears in the search results.
- ❖ We will also use Google Analytics in the production lane to determine whether or not our search function is being used to access branch pages in place of using the Google Search engine. This will be done because sometimes users will rather use the google.com search bar in order to find a page they want on a website rather than that website’s own search function.

- Ideas (Feature 14)
  - Users will be able to create, save, and update their ideas.
  - All ideas will either be (a) a seed or (b) a branch off of another seed or idea.
  - Ideas will have a title, tag(s), description, and pictures/schematics
  - In addition to the idea originator, other users will be able to up-vote an idea at most one time
  - ❖ The data structure representing Ideas within SeedIt will be very complex. Correctness of the Ideas feature will be ensured using Mocha unit tests on the database while in the stage lane. Many unit tests will be utilized to validate the Idea data structure. One such test will be to ensure that when another branch is created off of a seed, it will correctly contain its parent seed’s ID.
  - ❖ Karma will be used in the stage lane to make sure that the branch data is being correctly validated before being sent to the server by disallowing blank descriptions, titles, tags, and script injections in the text. The test will attempt to send seeds that violate one or more of these rules, and the GUI component for submitting the seed should not be activated until a valid seed appears in the view.
  - ❖ Protractor will be used to make sure the branch is correctly displayed on SeedIt. One such test is to make sure that when there are no parent branches for a seed, no link to a parent seed would exist. This will be done by programmatically creating a seed with no parent, then checking that the GUI component is not found on the page. Another feature to test is that the branch display will retain its size, even with larger pictures. This will be done by programmatically creating a branch, inserting a large 5000x5000 picture, and validating that the GUI component surrounding the picture is appropriately resized to fit inside the normal branch display.

Frontend UI Testing

Selenium

Overview

There are many advantages to test automation. Most are related to the repeatability of the tests and the speed at which the tests can be executed. Keeping in mind that this automation

is usually only valuable if there is a longer development cycle with many changes being made to the code base.
We will use selenium in the stage lane to test regular user tasks on the frontend. Selenium is a browser automation tool which will allow us to program tests to be run in a regression type testing fashion. The Selenium IDE is a Firefox add on and is recommended to do those types of tasks. Selenium IDE has a recording feature, which records user actions as they are performed and then exports them as a reusable script in one of many programming languages that can be later executed.

Where Selenium Would Be Useful
Selenium would be useful for all of the frontend test which can't be run by protractor or other testing software. Specifically we can use selenium to test filling out certain text boxes such as the search bar and then clicking on an area we know should have interaction after the search. Another great use of Selenium would be to test clicking on certain features to make sure they react as expected. Regression tests will be programmed in Selenium and added to our test suite after each feature is completed. This will ensure that all of our past features are still working even after our code base or development environment is changed.

Protractor

Protractor will also be used on the front-end in order to perform effective end-to-end-testing, since it is a part of an effective testing plan for Angular applications. Protractor tests will be written during the development phase of the iterative process as well as the staging phase. As the complexity of our application grows, more and more errors will go unfound if the proper E2E tests are not performed in order to find regression errors. Protractor is unique in that it allows you to test certain aspects of your GUI with the normal testing paradigm of comparing expected values with real values. This way, much less human testing is required and some returning defects will be discovered almost immediately upon resurfacing. The AngularJS site provides the following example of Protractor which demonstrates the real/expected paradigm:

```
  // Verify that there are 10 tasks
  expect(element.all(by.repeater('task in tasks')).count()).toEqual(10);

  // Enter 'groceries' into the element with ng-model="filterText"
  element(by.model('filterText')).sendKeys('groceries');

  // Verify that now there is only one item in the task list
  expect(element.all(by.repeater('task in tasks')).count()).toEqual(1);
```

Notice that Protractor allows the tester to programmatically interact with the units on the page and also allows the tester to quantify the correctness of the interaction's result. As seen

above, if the filter “groceries” was typed in and resulted in more than one item being in the list, this test would fail.

Using protractor would make it simple to test a larger Angular.JS application for errors. Moreover, we can write a test for each bug that we find so that an Angular.JS application with a defect will fail during the development phase. This will keep our application from regressing in other areas during development by keeping the developers aware whenever a bug has resurfaced.

Now that the advantages of using protractor have been clearly discussed, the specifics of what will be tested will now make more sense.

The app “SeedIt” contains many different views that are modular from one another. Routing is used so that the user can perform an action in SeedIt without ever refreshing the page. Each one of these modular pages will be subject to Protractor tests that are written in the development phase as a response to the following events:

1. Basic test cases that the developers think up naturally over the course of development
2. Tests written to react against current defects found

The first type of Protractor test will be made while the programmer is developing the application and making sure that it works on a basic level. For example, the list of SeedIt branches below the search bar should be populated with relevant results whenever a new search term is typed in the search bar. If any non-relevant result pops up in the search results, then there is clearly a defect.

The second type of Protractor test will be written to react against the defects the developers didn’t see for one reason or another. They will be written during the course of development on a responsive basis whenever a defect is discovered. Once a proper test has been written and fails the application in its defective state, only then will the programmer be “justified” in going to actually fix the defect in the code. This idea is meant to counteract the lack of complete developer clairvoyance.

Karma

Protractor is great for end-to-end testing, but for the testing plan to come around full-circle, a unit-testing framework will be needed. Karma fits this bill perfectly: it can unit-test the many different Angular.JS services in the code and verify their behavior, it can be used with the Jenkins CI server, and it can be configured to run tests in multiple browsers. This is why Karma will be used during the staging phase of our development process.

With Karma and the Jenkins CI server, we can test modular units of our AngularJS application and also be alerted when a newly inserted version control commit has failed one of our existing tests. This will allow us to be immediately alerted when a defect appears in our application that we've already written a test for.

Since Karma is primarily used for unit-testing, it will be invaluable for testing the Angular.JS app's services which gather JSON data from the main server. Consider the following Karma example, courtesy of Ben Drucker from Valet.IO:

We create the following Angular.JS service which will return a person object with the provided name in the constructor:

```
.factory('Person', function () {
  return function Person (name) {
    this.name = name;
  };
});
```

We verify that this service works by invoking the service's module, myApp, and injecting our previously defined service above into the variable Person. We can then test to ensure that the constructor that we made above actually assigns the name "Ben" to the attribute "name."

```
describe('Person', function () {

  var Person;
  beforeEach(module('myApp'));
  beforeEach(inject(function (_Person_) {
    Person = _Person_;
  }));

  describe('Constructor', function () {

    it('assigns a name', function () {
      expect(new Person('Ben')).to.have.property('name', 'Ben');
    });

  });
```

```
});
```

In this testing code snippet (which passes), we can verify that our constructor actually assigns the string parameter to the attribute “person.” We also see that we can easily organize our unit tests into units by using the describe() function as provided by Chai.js. We can also inject our dependencies using ngMock as seen in the module() and inject() functions. Last but not least, we can run these tests using the continuous integration server Jenkins whenever a new build is created.

Karma’s perfect for unit testing because it works with the continuous integration server, can effectively unit-test our angular modules, and allows us to run the code on several different browsers to verify cross-browser compatibility. This will be another invaluable piece to the puzzle for an effective testing plan of SeedIt. In order for a front-end commit to be pushed beyond the staging phase, it must pass the Karma unit tests.

Other UI Testing

Web UI has to be flexible since websites are viewed from different browsers and window sizes. Automated tests can't ensure that layout looks good in these varied conditions. Our web app should be responsive to the shape of the window that users are viewing it in. Responsiveness includes properly sized elements, text and sections. Also layout will have to change when there is not enough room to reasonably show elements in their original layout. We'll test our UI's reactivity by running it in the major browsers: Chrome, Firefox, Internet Explorer, Safari and Opera. We'll test it with different window sizes and zoom levels. We'll also test in different operating systems and mobile browsers. Other ways we'll test the responsive is with different amounts of text in our titles and descriptions, and different sized pictures attached to the ideas. This way we can verify that users will have a consistent experience no matter what browser they are using and what content a particular page holds. In the staging phase, the following process will be done to make sure the front-end handles various screen sizes correctly:

Process for testing responsive layout in desktop browsers:
1: Open SeedIt homepage on browser(Chrome, Firefox, Internet Explorer, Safari, Opera)
2: Maximize browser window.
3: Press Ctrl-0 to set zoom level to default.
4: Ensure that text, buttons, and pictures fit within the bounds defined by the UI design and in the proper location.
5: Using browser menus or Ctrl-+ and Ctrl--, set zoom levels to 50%, 75%, 90%, 110%, 125%, 150%, 175% and 200%. At each level, make sure all elements are properly sized and in the correct location according to the design.
6: Resize browser window to a width of 240 pixels.
7: Repeat steps 3 through 5.

8: Repeat steps 6 and 7 for widths of 320 pixels, 480 pixels, 768 pixels and 1024 pixels.
9: Navigate to a specific idea page.
10: Repeat steps 2 through 8.
11: Repeat step 9 and 10 for the new idea page, search result page, tag list page, about page, and login/signup page.
NOTE: This should be done on a computer monitor that can properly fit the different widths from step 8. It should be tested on Windows, Mac and Linux versions of the browsers when applicable.

Process for testing responsive layout in mobile browsers:
1: Open SeedIt homepage in mobile browser(iOS browser, Chrome for Android, Android browser, Firefox for Android, and Internet Explorer for Windows Phone)
2: Ensure that text, buttons, and pictures fit within the bounds defined by the UI design and in the proper location.
NOTE: This test should be done on iPhone 4, 5 and 6, Android Phones of variable sizes, iPads, iPad minis, Android 7 inch and 10 inch tablets.

The following tools can be used to imitate screen sizes and devices to make testing easier:

- Window Resizer - Chrome Plugin
- Window Resizer - Firefox Plugin
- Firefox Developers Tools Responsive Design View: https://developer.mozilla.org/en-US/docs/Tools/Responsive_Design_View
- Chrome Developers Tools Device Mode
- http://browsershots.org/ Automatically creates screenshots of a website running in different browsers at different versions and operating systems.

Frontend Integration Testing

Purpose

Since SeedIt is a complex piece of software, there will be multiple developers working on different modules of the front end. We want to be working on SeedIt in parallel so that no developers are left waiting on other modules in order to continue work. This can be accomplished by using stub classes to mimic the behavior of other modules that have not yet been completed. At some point, those modules will need to be integrated with one another. The purpose of our integration tests is to ensure a smooth integration process and minimal errors after integration.

Main Frontend Modules to Integration Test

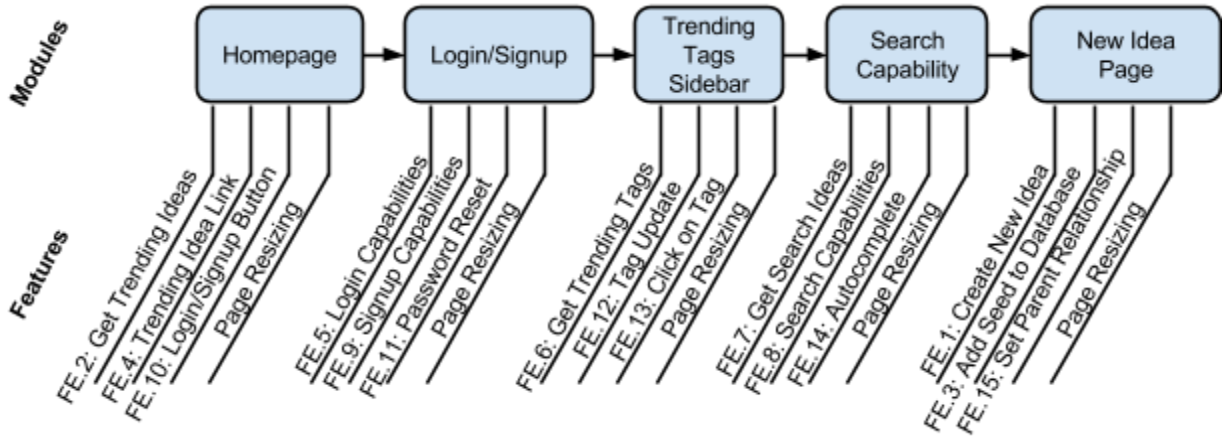
- Homepage
- New Idea page
- Trending Tags sidebar
- Search Capability
- Login/Signup page

Top Down Integration Testing

We will be pursuing a top down integration approach on each module during the staging phase. The following figure shows each module and the specific features associated to that module. Each feature will first be populated by a stub object to ensure that the module is working properly without any integration with the backend. Once all the features of a module are working then we will switch out the stub for backend service calls and data.

The arrows in the following figure indicate the order in which we will test each module and its integration with the previous integrated modules. This integration testing of modules will happen only after all the features have been fully integrated with the modules and tested.

Figure 1 - Frontend Integration Testing Plan



Feature Integration Testing

The following is a detailed explanation of what needs to be tested on each specific feature during the integration testing process.

Homepage Features:

- FE.2 - Get Trending Ideas -
    - Use Selenium to create a user story that access all pages and returns to the homepage after each page visit. Upon each visit to the homepage, ensure that trending ideas are up to date and correct.
    - Protractor and Karma Tests
        - Trending ideas are placed and styled correctly
        - Trending ideas are truly the top trending ideas
        - Voting scores are accurate and up to date.
- FE.4 - Trending Seeds have titles and links to related seeds

    - Use Selenium to create a user story that visits random seeds and then visits related seeds.
    - Protractor and Karma Tests
        - When a seed is added to the database it's title is correctly associated to the right seed.
        - A seed has the correct links to related seeds
- FE.10 - Login/Signup Button
    - Use selenium to create a user story that ensures that when the login and signup button is clicked the page changes to the login/signup view
    - Protractor and Karma Test
        - The Login/Signup button correctly redirects to the login/signup page

Login/Signup Features:

- FE.5 - User Login Capabilities
    - Use Selenium to create a user story that logs into a user account.
    - Protractor and Karma Tests
        - A user should be able to log in to their account
        - A person should not be able to log in to a user account without the correct username or password
        - When a user is logged in, there is a logout button available on all pages
        - The login/signup button is not viewable when a user is already logged in
- FE.9 - Signup Capabilities
    - Use Selenium to create a user story that signs up for a new user account
    - Protractor and Karma Tests
        - A new user should be able to sign up for a new user account
        - A new user cannot use the same username as an existing user
        - A new must have a password
- FE.11 - Password Reset
    - User Selenium to create a user story that resets a user's password
    - Protractor and Karma Tests
        - A user can reset their password by correctly answering security questions
        - A user cannot reset their password if the answers to the security questions are incorrect

Trending Tags Sidebar Features:

- FE.6 - Get Trending Tags
    - Protractor and Karma Tests
        - On page load, truly trending tags are loaded into the trending tag sidebar
- FE.12 - Tag Update
    - Use Selenium to create a user story that enters searches and changes views to evaluate what tags are in the sidebar
    - Protractor and Karma Tests
        - As the user searches, changes pages, and views seeds the tags on the sidebar update accordingly.
- FE.13 - Click on Tag

    - Use Selenium to create a user story that clicks on multiple tags in the sidebar to evaluate the correctness of the link
    - Protractor and Karma Tests
        - All tags on in the sidebar are clickable
        - When a tag is clicked on, the main view is populated with the top ideas associated with that tag

Search Capability Features

- FE.7 - Get Search Ideas
    - Use Selenium to create a user story that searches for multiple ideas
    - Protractor and Karma Tests
        - Get seed ideas that are relevant to the users search
- FE.8 - Search Capabilities
    - Use Selenium to create a user story that enters multiple different types of searches, valid and invalid, to ensure search functionality
    - Protractor and Karma Tests
        - Handle all types of searches and all types of characters
        - Handles script injection attacks
- FE.14 - Autocomplete
    - Use Selenium to create a user story that uses the autocomplete functionality of the search bar
    - Protractor and Karma Tests
        - As a user is typing in the search bar, valid search options are populated.
        - These search options can be auto completed
        - When a search option has been auto completed the user can press the search button to search that option

New Idea Page Features

- FE.1 - Create new idea
    - Use Selenium to create a user story that creates new ideas
    - Protractor and Karma Tests
        - Allow the user to create a new idea
        - Handles script injection attacks
        - Handles incomplete forms
        - Handles bad image links
- FE.3 - Add seed to database
    - Protractor and Karma Tests
        - Seed is correctly added to the stub database.
        - When integrated with the backend the seed is correctly added to the real database.
- FE.15: Seed/Parent Relationship
    - Use selenium to create a new seed and then access the parent of that seed if there is parental unit.
    - Protractor and Karma Tests

        - When a seed is added to the database, then the appropriate parent seed is correctly associated with the new seed.
        - If there is no parent seed, then the new seed is correctly labels as a root seed and there is not parent associated to the new seed.
- All features need to be tested to ensure that styling, position, and functionality works correctly with page resizing. These test can utilize selenium and window resizer.

Documenting Integration Errors

To ensure that all errors found while integration testing are properly taken care of, we have decided to document all errors with the thought of traceability in mind. All errors must be attributed to a feature that has a bug and be given an error number that is the same as the feature number. This will help our team to better understand the error and what is needed in order for the bug to be fixed.

Backend / Database Testing:

Purpose

The purpose of SeedIt backend / database testing is to ensure functional specification at the most granular level as well as at the application level. Unit tests done in the staging phase will help maintain that SeedIt operations are carried out correctly, with the correct preconditions and postconditions in place. Integration testing will maintain the application-level specification and functionality. Regression testing will sustain previous code as we continue to add new modules. The SeedIt backend / database test plan will help the SeedIt development team organize and define excellent user features that will become SeedIt's substance. This is how we'll know that our features are correct. While the problems that arise during software development are infinite, these tests will enable SeedIt to define and operate within a well-specified realm of expectation.

Scope

The SeedIt scope will consist of unit and integration tests as well as a regression suite involving tests as needed. Levels of testing will include: alpha, beta, and production release. The alpha level of testing will be entirely in-house, and will include specific unit, integration and unit tests. The beta level of testing will include a "semi-public" release catering to users of SeedIt. Beta testing will help the SeedIt development team to understand how people use SeedIt, and in turn, to write with those users in mind. The production release will consist of SeedIt being fully released to the public. Priority will be allotted to the minimal viable product as defined by this document. Tests involving additional features will only be added after the MVP is deemed complete.

Methodology

The respective leads of the front-end and back-end teams will run unit tests nightly during the staging phase, or make assignments to run them. Integration and regression tests will be run upon completion of milestones, or whenever it is deemed necessary by the team leads. See Figure 2.

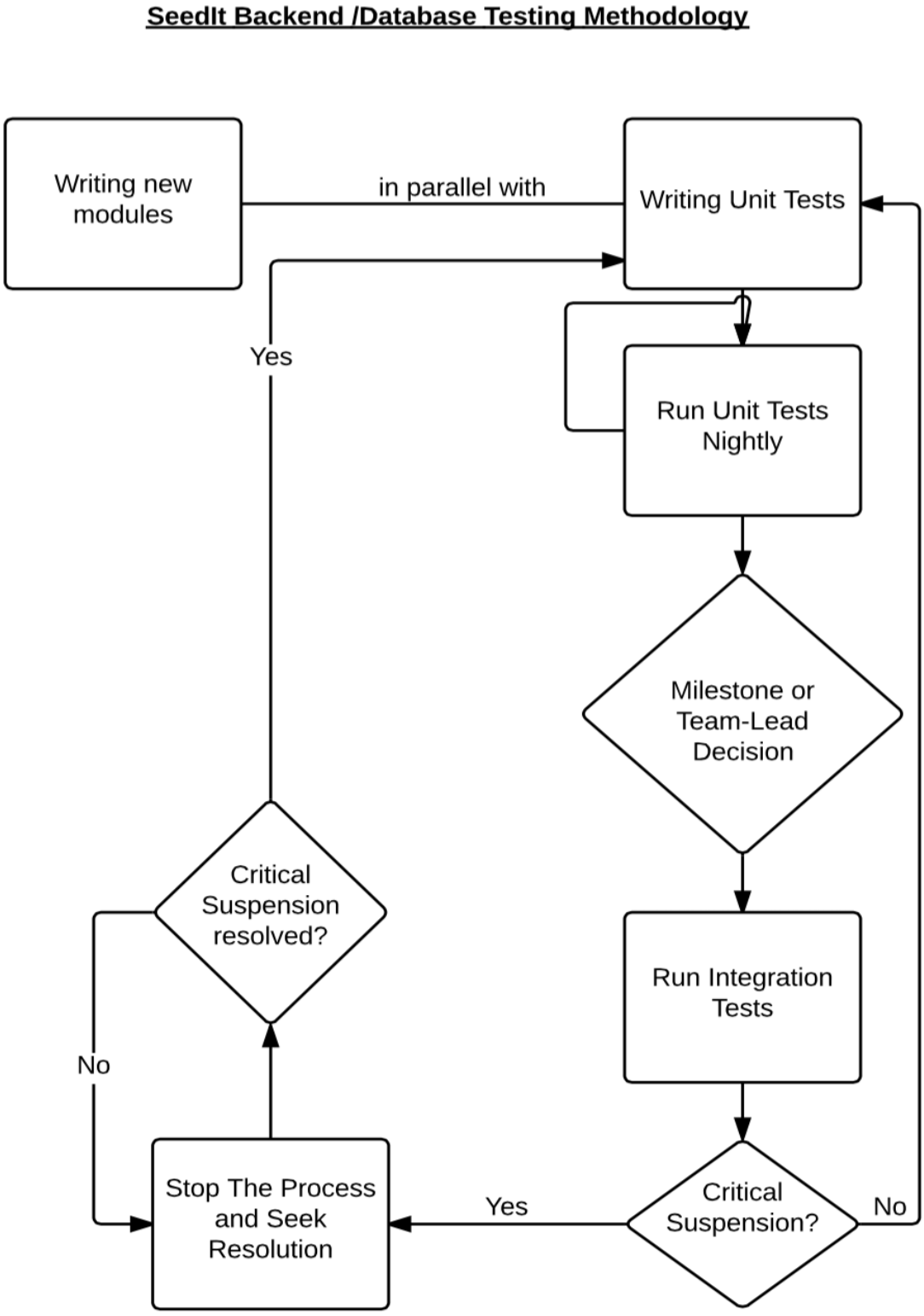


Figure 2. Flowchart of the backend / database testing process.

Requirements

SeedIt will use the Mocha framework to carry out unit tests. Each developer will have Node.js installed with npm. Hardware will include the developers' own laptops because we're poor. Team leads will carry out testing responsibilities unless otherwise assigned. All tests will be constructed with SeedIt features in mind, with well-defined criteria for passing and failure. The user experience will always be paramount.

Unit Tests Criteria for Pass-Fail:

The purpose and nature of SeedIt unit tests will be to keep operation of granular level logic with SeedIt features in mind. Each unit test will have preconditions and postconditions, specifying the bounds for input and functionality as defined by the respective feature. Boundary cases will check for invalid input. (In this case, both variables must be strings) Boundary cases will also include the empty string and maximum size string, etc.
Example:

```
/*Preconditions:
1. username and password must be supplied
2. username must be a valid username in the database
Postconditions:
1. The user is successfully authenticated
2. Access token is provided*/
testAuthUser(var username, var password)
{
        //execute test
}
```

Integration Tests Criteria for Pass-Fail:

Integration tests will include several modules together to ensure correct operation and correct data transfer as SeedIt features are executed. Again, the user experience is always paramount. Since several parts of the system will be tested, representing various features, several individual tests could be used. Integration tests criteria for passing will assume no critical breakage of system functionality. Integration tests criteria for failing will assume critical breakage and will prompt suspension until resolved.
Example (Run all tests in line and check data transfer at each step):

```
registerUser()
authUser()
createIdea()
addTagToIdea()
up-voteIdea()
getUserIdeas()
updateUser()
getAllIdeas()
getAllTags()
deleteTag()
getTagIdeas()
```

Schedule:

Unit tests will be run nightly on commits until the commits graduate to the production phase. Integration tests will be run after the March 18, March 31, and April 13 milestones, or upon team-lead decision. Unit tests, integration tests, and bug-fix tests will be ran as a regression suite of tests as needed.

UX

Purpose

Our the chief aim as a UX team is to continuously learn how the collaborative ideation process of SeedIt can be improved to help us move closer to our vision of inspiring people to rethink everything. We believe we can facilitate our innovation process through observing patterns in feature usage and studying how our users interact and respond to disruptive innovations that lead to superior solutions. Through discovering and analyzing these trends, and working alongside SeedIt users, we can improve our tools to capture,communicate and evaluate ideas that spark new veins of thinking.

Mantra

We are the idea hub

Philosophies that guide our UX Discovery Process

- Ideas are fun, structuring ideas is painful.
- Disruptive and key ideas can come from anyone.
- Ideas are fleeting, therefore efficiency is paramount.
- Ideas will thrive in an unabashed and playful environment.

Most of our studying will take place on the production side of our testing process. The richest information that will lead our UX team to formulating the right questions and making breakthrough innovations for our own tools and processes will come directly from studying the trends in the ideation trees. As we run into phenomena and anomalies that we can't explain with the numbers in our database we'll have to develop additional metrics that we can gather from user activity to help us propose new theories in user behavior.

How the users will experience SeedIt and where we will gather our metrics fall into these categories:

1) Creating
2) Browsing
3) Engaging
4) Evaluating
5) Communicating
6) Sharing

These categories will help us evaluate the trends that lead to pivotal and successful ideas while also helping us to create metrics to discern how changes to our features have stimulated or diminished activity in any one of these key areas.

Other important factors and data that our team will analyze will come from looking at how users are trafficking into and out our site and what third party sites they may have been using to help them formulate ideas and where they might be going to use these ideas. Using Google Analytics we can study these trafficking patterns to discover what features we may be missing that is causing our users resolve to using third party sites and tools. Knowing to what end they are using this information will play a key role in helping us decide what partnerships we can develop, what incentives our users and where we can focus our marketing efforts to attract in new contributors.

Code Review Process

On our team we are going to combine the two most common code reviewing methods as a requirement to graduate a commit to the staging phase: over-the-shoulder and email pass-around. As a small “startup” type company, we are often coding together. We are also mostly working in technologies that are new to us. In this type of atmosphere the over-the-shoulder code review is extremely easy to execute and allows for great learning experiences. In-person reviews are typically the most informal and allow for the reviewer and author to simply blurt out thoughts and ideas as they come. This can create an atmosphere where learning and sharing between developers comes easily and naturally. It also makes it very simple for the author to explain difficult or strange pieces of his code. Rather than attempting to email back and forth about a tricky line or two of code, the pair can discuss what is happening and the reasons it needed to be done that way.

Although we will tend to perform over-the-shoulder code reviews, there will be times when no one is around or available to perform the code review. In these cases we will use the email pass-around method of code reviewing. To do so we will use GitHub’s comments and RSS feed to alert each other of commits that need to be code reviewed. The reviewer can then look over the diffs provided by GitHub and respond in the comments with any suggestions or questions. Each member of the team will write code on their own branch. This way we are not committing unreviewed code into our code base. Once the code has been reviewed then the lead of the team will merge their branch into the development branch.

Code reviews should be done by the lead developer on your team. If you are the lead developer, then you may choose one of your team members to review your code. During this code review it is important to determine if a member of another team is needed. This could be the case if changes to the backend are being made that might affect the way the frontend interacts with the server/database. In that case, a member of the frontend team should also be present or notified through GitHub of a need to review a commit.

Code Review Rules to follow:

- Review fewer than 200 - 400 lines of code at a time
- Review should last no longer than 1 hour
- Authors annotate source code before review begins
- Foster the attitude that finding bugs is GOOD!
- Do some form of code review for every commit - knowing someone else will look at your code will make you write better code