Devoir

Test Plan and Specification

March 21, 2015

Project Team

Candice Davis
Alan Dayton
Brady Kelley
Cory Kirkland
Skyler Mitchell
Adam Nelson
Brent Roberts
Seung-hee Yang

Table of Contents

Introduction

This document is a high level overview defining our testing strategy for Devoir, a multi-platform homework tracking application. To test Devoir we plan to take each of the apps functional requirements[-1] and test them to ensure proper functionality on both the front and back end. In this document we will also discuss the general testing strategies we will use.

Roles and Responsibilities

Each development team (server, mobile, and web) will be responsible for testing their code base. We will also have a specific Usability Team that will be responsible for for conducting and recording the usability tests.

Server Team	**Mobile Team**	**Web Team**	**Usability Team**
Alan Dayton	Candice Davis	Cory Kirkland	Candice Davis
Skyler Mitchell	Brady Kelley	Seung-hee Yang	Brady Kelley
	Adam Nelson		Seung-hee Yang
	Brent Roberts		

Acceptance Criteria

In order for the software to be considered "ready to ship", all test cases for the current milestone must pass. See the *Test Schedule* section for details on which test cases must pass for each milestones. In addition to the tests cases assigned to each milestone, relevant usability, stress and performance tests will be conducted prior to acceptance. Later milestones must pass all tests of a previous milestone to be considered "ready to ship".

Testing Overview

Usability Tests

We will be testing the usability of both the mobile and web app with live users tests. After completing initial in house alpha test. We will invite students who are not familiar with the project to complete the usability test.

We will conduct two types of test:

1. Blind test - user is given the app without any prior knowledge of how to operate it. User is given a set amount of time to use the app, then report on usability, functionality, UX/UI.
2. Requirements test - user is given requirements and rates how effectively app meets those

Goal: users should be able to use the web app and mobile app with ease. We expect a high SUS[A-2] (System Usability Scale) score due to the efficient graphical user interface.

Preparation:

- Prepare one laboratory equipped with a PC (or mobile phone) with Internet
- Scenario: users are given a list of instruction from the proctor. The instruction asks testers to take a role as students who are registered in CS101. The following is a series of tasks for them to do as students:
  1. They need to sign-in with their Gmail account.
  2. They are asked to import the calendar information to Devoir Web app.
  3. They can see their homework schedule in 3 different views.
  4. Some homework were changed by the instructor, so students are asked to update their calendar.
  5. Some homeworks were removed, so students are asked to update their calendar.
  6. They need to sign-out with Devoir Web app.

Plan:

1. Recruit people with various backgrounds. To avoid bias, exclude those who are already familiar to the system.
2. Invite them to the testing environment.
3. Conduct the scenario.
4. At the end of scenario, users are required to fill out survey questions.
5. Calculate SUS score.

Security Tests

Server

The server will require a lot of tests to protect against security threats. Tests will be needed to ensure that our server properly handles invalid requests and does not slow down in performance. The same

stress tests for get requests described below will be used with invalid requests to see if the server can handle it. This will protect against denial of service attacks, as unlikely as they are. More importantly, it will protect us against badly formatted requests if someone decides to build their own client that interfaces with our server.

Unit tests must be done to ensure that all secure endpoints are locked down. No one should be able to access any endpoint without first logging in through Google, and every request they send should be accompanied by a token so that we know the request came from them. Every endpoint (except for the login and some static endpoints) will be hit without this token to ensure that a 403 error is returned.

Database security is also essential to consider. Manual testing will be done to ensure that no one without a password can access the database. Also, manual testing will be done to ensure that the user created for the server's use can update and read from the database, but cannot alter the database or any of the tables. Tests will also be done to ensure that SQL injection is properly handled by the third party module we are using to interact with the database.

Mobile / Web
Both web and mobile devices will rely heavily upon the security of the server. We will use Google OAuth to authenticate users. Without authentication we must not let the user into our application. We will test that invalid authentication credentials result in denial of access to our service, as well as that valid credentials properly allow users to view and edit their task list.

Stress Tests

Server
Though we do not have a large client base right now, we need to test that our server will be able to handle a large number of users. Since we do not have a large number of resources or computers at our disposal, this can be simulated by a large number of requests. Both the scalability of our server and of the database must be tested.

To test the scalability of the server and database, we will set up several computers to repeatedly send requests to the server. We will aim to send 1000 get requests per second to the server over the course of a minute and ensure that the server does not crash and that it finishes within 15 seconds after the minute is up. Then we will test the heaviest endpoint, which is importing an iCalendar feed. Since this requires at least dozens of inserts into the database, this will test more thoroughly the scalability of our interactions with the database. We will aim to send 100 requests to import an iCalendar feed each second over the course of a minute, and check that the server does not crash and that it finishes within 30 seconds after the minute is up.

Mobile

We will want to ensure that the mobile app can handle large amounts of data. We will programmatically create large amounts of simulated data such as tasks and feeds to ensure that our list views can handle large amounts of data. If we do find that there are limitations to the number of events we can handle per feed, then we will warn the user if we believe this might be an issue for them.

As part of testing that we can synchronize the state of events after loss of internet availability we will mark many tasks as complete and add notes to some tasks while offline. Further, we will mark some tasks as hidden. We will do this on a large scale to ensure that the local storage is enough to be able to remember everything that must be updated on the server side once internet connectivity is restored.

Performance Tests

Server

All requests should be handled in under 500 ms, except importing an iCalendar feed, which should be handled in under 1000 ms. Each endpoint will be tested with a load of 100 requests per second to see if they meet the time requirement.

The core benefit of NodeJS as server-side language is that it is based on evented, non-blocking I/O. The language is based on V8 (Javascript engine for Chrome) known for its high performance. Thus, higher performance is expected than the popular Apache server. We will build an apache server with the same functionality and test the difference in requests handled per second and the memory and CPU usage for each server.

Mobile

On both mobile platforms, performance tests will be conducted to ensure the users has a smooth experience throughout the app. Even though certain operations may take "longer" to finish (ex. Adding an iCal feed), the app needs to appear to run without delay or slow downs. The user needs to continually feel that the app immediately responds to input, regardless of the action taken.

Main focuses of mobile performance tests:

- Can the app handle large data sets?

  Procedure: Add at least 20 large iCal feeds to an account. The goal of adding these iCal feeds is to populate the task view with many tasks from multiple feeds. Make sure the task view moves smoothly from one day to the next (whether swiping on Android or scrolling on iOS). Make sure marking and unmarking tasks is smooth and unhindered. Also, test to make sure that using the datepicker to jump to a day happens quickly and smoothly.
- Does the data stay in sync?

Procedure: Put the device into airplane mode to force no connection. Mark multiple items as completed, hide multiple items, etc. Log out of the app. Restore connectivity of the device. Log back into the app and allow the app to sync with the server. All changes made while the device had no connection should be preserve and synced to the server.

- Can the user log in and out repeatedly without losing user data?

  Procedure: Log in as a new user to the app. Add multiple iCalendar feeds. Manually add a course and a few tasks. Mark off multiple tasks on multiple days. Take note of which tasks were marked complete. Log out and then back into the app. Compare the state of the data before logging out to the current state.
- Force close the app during important procedures to see if data integrity is maintained.

  Procedure: Add an iCalendar feed to the app, while downloading the feed force close the app. To force close on Android, press the recent app hardware button, swipe left on Devoir to force close. To force close on iOS double press the home button, long press Devoir and tap the X button to close. In each of these cases if the app was force closed before the full download, no information should be saved to prevent data corruption. In the case of updating an already added iCalendar feed, updates should be discarded. Adding and updating iCalendar feeds should be done in one transaction and therefore not have partial saves due to a force close.
- Can the user log into multiple devices at the same time?

  Procedure: Log a user into multiple devices. For example: Android phone, iPhone, and the web. Changes made on one platform should be reflected on the other platform after the app has synced. If a user migrates from Android to iOS or vice versa, then all data added by the user should be available.
- Is data integrity maintained when user has multiple logins?

  Procedure: Similar to previous test, except focus should be made on making sure data is synced correctly across each platform.
- Run all above tests on range of hardware

  Procedure: All tests above should be run on hardware specs given below.

  For iOS:

  Models: iPhone 4S or later.

  iOS version: 6.0 or above.

  For Android:

  Android version: 4.0 or above

  CPU: 800 Mhz or above

  RAM: 800 MB or above

  Screen Resolution: ldpi-xxhdpi

Web

The Web app test whether the functions provided by the server works well on web front-end. The test focuses are:

- Can same user running the page on multiple tabs/browsers simultaneously?

- Does the information sync across all open instances?

Integration Tests

Server

Test that each platform can correctly authenticate with Google and that each subsequent request to the server contains the appropriate token and the server accepts it. Test all of the endpoints from each platform to make sure the return the correct responses and check the server database to ensure its contents are correct.

Mobile

Similar to JUnit new integrations must pass all tests before and after integration. We will also need to focus manual testing efforts on ensuring that the mobile device can communicate correctly with the server once we connect the two ends. This will be the most important integration test we will have to handle.

Web

Any changes will have to pass all integration and unit tests in order to be added to the repository. These will be particularly important as new functionality is added. A lot of this will be focused on the UI working properly.

Platform Tests

Android

Devoir will be tested on many form factors representing the large majority of android devices. This includes: High to Low end cpu's and ram, High to low resolution screens, Phablet sized phones…

Current hardware requirements: Android 4.0 or higher.

Tablets will be introduced later, but a similar procedure will be conducted.

iOS

Devoir will be tested on all iPhone devices model 4S and above. Tests will be run on all operating systems above 6.0. At this point Devoir will not support tablets, tablet integration may be implemented at a later time.

Web

The Devoir web app will be tested on the following web browsers:

- Chrome
- Firefox
- Safari
- Internet Explorer
- Opera

Unit Tests

Unit Testing is done at the source or code level for language-specific programming errors such as bad syntax, logic errors, or to test particular functions or code modules. The unit test cases shall be designed to test the validity of the programs correctness.

Server

Unit tests will be written for each public function in all of the model/database access classes. The remainder of the server will be tested from end to end by sending requests to every endpoint and validating the responses and the state of the database. See the functionality tests for descriptions of specific tests.

Mobile

Unit tests

All classes have Units test. All tests must pass before code is committed to repository. We will not focus our efforts on code coverage, as there are many getters and setters which do not need to be heavily tested (if at all). Instead we will ensure that all functions that do the "heavy lifting" are well covered by our automated test cases.

Web

All classes have Units test. All tests must pass before code is committed to repository. For specifics, see the individual requirements to see how we are going to use the unit tests to ensure that the web app is up to snuff.

UI Tests

Mobile

Mobile UI tests we be run manually in both alpha and beta testing. We will be checking that each UI element behaves as expected.This will also be part of platform testing; ensuring that all UI elements are displaying correctly on the various devices.

Web

We will use selenium tests to automate the tests on the user interface. These will be able to repeat the same sequence of clicks and inputs in order to test that the proper things happen.

See specific requirements for details of how each part will be UI tested

Functionality Tests

MVP (Version 1.0) Tests

1.0.1 Import from iCalendar feed

Server

Relevant Endpoints:

POST /api/courses

Parameters: userID, name, color, visible, icalFeedUrl (optional)

Imports the course and task information corresponding to the given iCalendar feed URL. If no iCalendar feed URL is provided, a simple course is created. See section 1.1.1.

Overview

When the endpoint above is hit the server will request the iCalendar feed data and parse and store it in the database. This will require unit tests at the level of the endpoints and at the database access classes.

Unit Tests

The endpoint unit tests will include valid and invalid user IDs, valid and invalid iCalendar feed URLs, iCalendar feed URLs that have already been imported, and different combinations of null values for the other parameters.

All of the interaction with the database goes through database access classes. Each of these database access classes have create, read, update, and delete functions. Unit tests will be used to validate each of these functions and also that the server is connecting with the database properly. The unit tests will include both valid inputs and invalid inputs. The invalid inputs will consist of incorrect data types and null values to make sure that errors are handled properly. The invalid inputs will also include SQL injection attacks in order to make sure that the third party module we are using properly handles SQL injection attacks.

Mobile

When creating a new course the mobile app can take an iCal feed URL as input. The app will send that url to the server who will in turn respond with all that feeds iCal events in the format of tasks formatted as JSON. The mobile app should then parse the JSON into task objects and insert them into the database.

Unit Tests

We will use unit tests to make sure that the mobile app is properly formatting the request that it sends to the server. Unit tests will also be used ensure that the mobile app is parsing the response from the server into task objects and inserting those objects into the database properly.

UI Tests
The UI will be tested to make sure that when creating a course there is an input field for the iCal feed URL. After the course has been created we will check the task list in order to see if the tasks from the iCal feed have been properly added to the task list.

Web
Unit Tests

We will need to test that a user can add a valid iCal feed and import it properly. The server is the one actually doing the importing, so these test will be to ensure that the proper thing happens on the web end both in submitting the info and receiving the proper response. Tests will include things like:

add feed
add multiple ical feeds
start adding a feed then stop, then try to add the same feed
add a feed, logout, login, try to add same feed, try to add different feed

We will also make sure the webpage handles invalid add iCal submissions. Test will includes things like:

add same feed multiple times
try adding custom feed with iCal feed properties
try adding invalid iCal feed

UI Tests

We will test adding iCal feeds. When properly added, the page adds a course to the list of courses. When the input is invalid, it will not add the course, but will prompt the user to input correct values

1.0.2 Edit/Delete Courses

Server
Relevant Endpoints:

PUT /api/courses/{courseID}
Parameters: userID, name, color, visible, icalFeedUrl (optional)
Updates the specified calendar.

DELETE /api/courses/{courseID}
Parameters: userID
Deletes the specified calendar.

Unit Tests

Testing the put request will involve unit tests with valid and invalid course IDs, valid and invalid user IDs, and different combinations of null values for the other parameters. Special consideration will be given to testing the iCalendar feed URL. If the URL is the same, only the other parameters should be updated. If it is different, the old calendar data should remain, but the new calendar data should be added.

Testing the delete request will involve unit tests with valid and invalid course IDs and valid and invalid user IDs. The database will be checked to ensure the course and all tasks associated with it have been properly dropped. To make sure the endpoint is robust, tests will be performed to check that multiple deletes of the same course are handled correctly. Tests will also be performed that will ensure a user can only drop a course that belongs to him/her.

Mobile

The mobile app should allow users to edit any course. The fields of the course that are editable are: name and/or color. A user can also delete a course.

Unit Tests

We will use unit tests to make sure that our database can properly handle updating an existing course. To test this we will execute the methods on the database access classes that are responsible for updating a course and then check the database to ensure that the proper changes were made. We will also try try to update fields which are not editable to make sure that our database access classes properly handle those cases. We will also execute methods for deleting a course and then check the database to make sure that the course was actually removed. When the course is deleted all tasks associated with that course should also be removed from the database.

UI Tests

On the client side a course can be deleted by pulling the course cell in the table view to the right. This will slide the cell and show a delete button beneath it. We will attempt this action and make sure that everything functions properly and that the course is gracefully removed from the table when it is deleted. We will also test editing a course by sliding the cell to the right and pressing the edit button which should take us to the edit course page. On that page information about the course will be displayed, we should be able to edit this data and those changes we make should be saved to the database after leaving the page. After the course has been deleted we will check the task list to ensure that all tasks associated with the removed course have also been removed.

Web

Unit Tests

We will test both adding a course and removing a course. Tests will include things like adding a valid course, then deleting it. We will test persistence of course edits by editing or deleting a course, logging out and back in, and then checking that the course's changes are still in effect. We will also

edit the different parts of the course (ie. name, color, iCal feed) with both valid and invalid edits to ensure that only valid edits are actually executed.

UI Tests
We will create tests to check that color and course name changes properly. We will check that deleted courses don't show up and that no tasks from the deleted course appear on the calendar or the task list. We will also check the persistence of changes over different login sessions.

1.0.3 Create Tasks from iCal Event

Server
Relevant Endpoints:

POST /api/courses/{courseID}/tasks
Parameters: userID, name, description, startDate, endDate, completed, visible,
Creates a new task belonging to the specified calendar.

Unit Tests
Tests will be performed to ensure that attempts to create a task with an invalid courseID or userID are rejected. Another test case will be to ensure that a request to create a task for a course that does not belong to the user is rejected. The remainder of test cases will be for valid input, including various combinations of null values.

Unit tests will also be used to ensure that when a user correctly hits this endpoint that the user_last_updated field in the database is updated correctly.

Mobile
The mobile app should be able to send a request to the server for getting all new tasks from the specified iCal feed. The response from the server will contain all the new tasks in JSON format.

Unit Tests
We will use unit tests to ensure that the mobile app is properly formatting and sending the request to the server. Unit tests will also be used to make sure the mobile app is receiving a response and parsing the JSON into task objects. After the request has been made the database will be checked to make sure that the app properly updated the database with the new tasks.

UI Tests
We will visually confirm that the UI is functioning properly. To do this we will refresh the iCal feed and make sure that the new tasks are being properly displayed in the task list.

Web

Unit Tests

We will test that the webpage angular framework can properly parse tasks from json into tasks that can be displayed on our page. We will test that this works for multiple feeds and that they show all respective tasks properly

UI Tests

Display of tasks in the calendar modes is being done with a third-party calendar module. We don't need to test the calendar's functionality extensively. We just need to test that it displays how we want it to display. We will test that tasks display properly in the calendar. We will also test to make sure that multiple tasks on same day from the same feed and from different feeds display how we want them.We will also ensure that tasks that span multiple days are properly displayed

1.0.4 Edit/Delete Existing Task

Server

Relevant Endpoints:

PUT /api/tasks/{taskID}
Parameters: userID, name, description, startDate, endDate, completed, visible
Updates the specified task.

DELETE /api/tasks/{taskID}
Parameters: userID
Deletes the specified task.

Unit Tests

Testing the put request will involve unit tests with valid and invalid task IDs, valid and invalid user IDs, and different combinations of null values for the other parameters. Test cases will also ensure that the user_last_updated field is properly changed when users hit this endpoint.

Testing the delete request will involve unit tests with valid and invalid task IDs and valid and invalid user IDs. The database will be checked to ensure the task has been properly dropped. Again, tests will be performed to check that multiple deletes of the same task are handled correctly. Tests will also be performed that will ensure a user can only drop a task that belongs to one of the user's courses.

Mobile

The mobile app should allow users to edit any task. The fields of the task that are editable are: name, course and/or due date. A user can also delete a task.

Unit Tests

We will use unit tests to make sure that our database can properly handle updating an existing task. To test this we will execute the methods on the database access classes that are responsible for updating a task and then check the database to ensure that the proper changes were made. We will also try try to update fields which are not editable to make sure that our database access classes properly handle those cases. We will also execute methods for deleting tasks and then check the database to make sure that the tasks were actually removed.

UI Tests
On the client side a task can be deleted by pulling the task cell in the table view to the right. This will slide the cell and show a delete button beneath it. We will attempt this action and make sure that everything functions properly and that the task is gracefully removed from the table when it is deleted. We will also test editing a task by pressing on the task cell which should take us to the edit task page. On that page information about the task will be displayed, we should be able to edit this data and those changes we make should be saved to the database after leaving the page.

Web
Unit Tests
We need to test that we can edit task information like name, course and due date.We will try valid cases: changing a task from an existing course to different existing course, and form a course to the same course.We will also try invalid cases: bad names, changing to non existent courses, due dates that are impossible. We will also test the ability to delete a task and trying to delete a task multiple times.

UI Tests
We will test that the edit task view appears on double-click or right click followed by clicking "edit" prompt. We will test the following:

- Can cancel edits before finalizing them
- Editing due date properly updates task in calendar
- Editing course changes task's course color and info
- Name changes display properly
- A deleted task no longer appears in calendar
- All changes persistent between sessions

1.0.5 Marking Tasks as Complete

Server
Relevant Endpoints:

PUT /api/tasks/{taskID}
Parameters: userID, name, description, startDate, endDate, completed, visible
Updates the specified task.

Unit Tests
The unit tests will be the same as in 1.0.4. A specific test case will be devoted to checking that the completed property of a task is updated properly along with the user_last_updated field in the database.

Mobile
Through the mobile app the user should be able to mark tasks off as complete. When the task has been marked as complete it should be removed from the section of incomplete tasks and moved into a section of completed tasks which will be displayed at the bottom of the task list. This change should be consistent throughout all the apps.

Unit Tests
We will use units tests to ensure that our database access classes are updating our database properly. We will also use unit tests to send requests to the server and ensure that our requests are formatted properly and that we are receiving the responses from the server. Unit tests will be be executed using both valid and invalid data to make sure that our database access classes and server communication responds appropriately. We will also run unit tests while the app is offline to make sure that the application handles updating its own database and then sending a request to the server once it comes back online.

UI Tests
We will use the mobile apps to mark off tasks as complete and visually confirm that the UI is properly updated. This will involve marking tasks off as complete and checking to see if they are removed from the list of incomplete tasks and added to the list of completed tasks. In addition to updating properly we want to ensure that they UI is updated in a smooth and quick fashion. UI tests will also be run while the app is offline to make sure that the device currently used it updated immediately and that after the device reconnects to the internet any instance running on a different device becomes synced.

Web
Unit Tests
We will test that tasks can be marked as complete and that tasks can be toggled between complete and incomplete.

UI Tests
We will test clicking the completion box for a task works. When clicked, the completed task needs to change color. Unclicking a completed task changes its color back to its proper course color.

1.0.6 Task List View
Server
Relevant Endpoints:

GET /api/courses
Parameters: userID
Returns a list of the user's courses.

GET /api/courses/{courseID}/tasks
Parameters: userID
Returns all the tasks for a particular calendar.

Unit Tests
Testing will involve putting various test fixtures in a test database. These fixtures include:

- an empty database
- a database with no courses belonging to the user
- a database with a course belonging to the user but no tasks
- a database with a course belonging to the user with tasks
- a database with multiple courses belonging to the user but all with no tasks
- a database with multiple courses belonging to the user and some have no tasks
- a database with multiple courses belonging to the user and all have tasks

Unit tests will ensure the proper outcome in all of these cases. The security will also be tested to ensure that only the courses belonging to a user are returned and tasks are only retrieved if the task belongs to one of the user's courses.

Mobile
The task list view should display all of the tasks for all the courses for the logged in user.

Unit Tests
We will use units tests to make sure that tasks are properly being pulled from the database. To do this we will try to retrieve tasks from the database through the database access classes. We will run these tests on a populated database and an empty database.

UI Tests
To test the UI we will simply navigate to the task list view and confirm that it is displaying all the tasks as expected. We expect that all the tasks for all courses are displayed and sorted according to their due date.

Web
UI Tests
We will test that a user can toggle between calendar views and task list view. We will also check that completed and uncompleted tasks are properly displayed and that users can edit tasks in the task list view.

Version 1.1 Tests

1.1.1 Create a Course Without an iCalendar Feed

Server

Relevant Endpoints:

POST /api/courses

Parameters: userID, name, color, visible, icalFeedUrl (optional)

Imports the course and task information corresponding to the given iCalendar feed URL. If no iCalendar feed URL is provided, a simple course is created. See section 1.0.1.

Unit Tests

For a description of the tests for the database access classes, see section 1.0.1. Unit tests will simply be to ensure that the course can be created properly when no iCalendar feed URL is provided. Invalid and valid user IDs will be tested, as well as various combinations of null values for the other parameters.

Mobile

Users should be able to add a course manually. These courses are not linked to an iCalendar feed, and thus will never be updated except directly by the user themselves. These courses will look and feel like the courses added through an iCalendar feed. Functionally, they should be no different than iCalendar courses.

Unit Tests

We will perform unit tests to ensure that courses can be created and saved correctly to the database. This will include making sure that requests are properly made to the server after a user creates a new course. Valid data will be used to make sure that data is saved then subsequently queried to test that data objects are populated correctly from the database. Invalid data will be used to test against saving unexpected/invalid input. We will also run tests offline to make sure the app properly syncs with the server when a connection is restored and all data is kept intact.

UI Tests

Feedback given to user on invalid inputs (Ex. empty string, whitespace only for course name).
User can select from a variety of colors
User must meet input requirements before course is created.

Web

Unit Tests

Tests will create a valid custom course without an iCal feed associated with it. We will also attempt to create a custom course with bad information such as having a valid iCal feed associated with the course, having an invalid iCal feed associated with it and general bad inputs all around.

UI Tests
Our tests will check to ensure valid inputs before allowing user to submit, prompting users where the input needs to be changed when invalid. We will also test to ensure that the created custom course shows up properly and persists.

1.1.2 User Creates a Custom Task

Server
Relevant Endpoints:

POST /api/courses/{courseID}/tasks
Parameters: userID, name, description, startDate, endDate, completed, visible,
Creates a new task belonging to the specified calendar.

Unit Tests
Tests will be the same as described in section 1.0.3, with particular attention to ensure that when a user correctly hits this endpoint that the user_last_updated field in the database is updated correctly.

Mobile
User can manually add a task to a course. An "add assignment" view is presented to the user. The user can give the task a name, description, due date and assign it to a course. The task will then show up in the task view on the date specified by the user. The task will assume the color of the course it was assigned to.

Unit Tests
Task objects will be tested. Course objects will be tested to ensure that they contain the correct tasks.

UI Tests
Tests will include making sure that proper feedback is shown when invalid data is entered by the user. (Ex. empty name, whitespace only name, no due date,etc.) Test to make sure a user cannot add more than one of the same task at the same time (double-click protection).

Web
Unit Tests
We will test that a user can create tasks for a custom course and can create tasks for an imported iCal course. We will also test bad stuff like creating duplicate tasks and invalid inputs or no inputs.

UI Tests
Tests will ensure that tasks are only created when valid and that tasks show up with proper info. We will check that the created tasks display in their proper courses and that the created tasks are editable just like imported tasks.

1.1.2.a Name, Due Date, Course, Description

Server

Unit Tests

Unit tests will include checking valid and invalid input, including incorrect date values and course IDs that don't match any existing course.

Mobile

Tested in section 1.1.2.

Web

Unit Tests

We will test both valid inputs and invalid inputs.

UI Tests

We will test that the webpage only accepts valid inputs. Tasks can't be part of nonexistent courses. Text fields with bad input will turn red and a prompt will come up when given invalid inputs.

1.1.3 Sort Tasks by Class or Date

Server

Sorting will be done on the client side, so no tests are required.

Mobile

The task list view has the option to be filtered by course. This will still keep the tasks ordered by date but will only show the tasks for a specific course.

Unit Tests

Unit tests will be used to test the database access classes and ensure that they are properly retrieving tasks based on the specified course. We will run tests using courses that have tasks and also courses that do not have any tasks. We will also try to retrieve tasks for courses that do not exist to ensure error checking is working properly.

UI Tests

To test the UI we will navigate to the task list view and then filter the list by course. We will then make sure that only tasks for the specified course are being displayed. We will also make sure that tasks are still being sorted by date no matter which courses are being viewed.

Web

Unit Tests

We will test that the display list only has selected course(s) while the course list has all the courses. The display list will change which classes are sorted. In both calendar and task list, all tasks are always displayed by date. We will test to ensure that this is the case.

UI Tests
We will test that the sorting will change which courses are display, ensuring that in both task list and calendar view only tasks from those courses are displayed.

1.1.4 Flag Task if Past Due

Server
Flagging will be done on the client side, so no tests are required.

Mobile
Tasks that are past due should be marked as overdue with a red label. They should also be displayed at the top of the task list in a designated overdue section.

UI Tests
We will navigate to the task list view and ensure that overdue tasks are marked off with a red label and that they are being displayed in the overdue section at the top of the list.

Web
Unit Tests
We will test this by creating a task that is past due and checking if its flag gets set for being past due.

UI Tests
Tests will ensure that in the calendar view, past due tasks change to red. In the task list view, past due tasks change to red and go to the past due list.

1.1.5 Completed Tasks Move to a Different Section

Server
This is done on the client side, so no tests are required.

Mobile
A user can press a checkbox on the task cell, this will mark the task off as complete and move the cell to the completed section at the bottom of the list.

Unit Tests
The backend for marking tasks off as complete was already tested in section 1.0.5.

UI Tests
We will mark tasks as complete by pressing the checkbox on the task cell. We will make sure that after we mark it off the task is removed from its current position in the list and inserted in the completed section at the bottom of the list. Any overdue marking from the task should also be removed.

Web
UI Tests
Tests will ensure that in the task list view, marking a task completed properly moves the task to the completed tasks section. Also, marking a completed task as incomplete moves it back to proper spot in the uncompleted tasks section

1.1.6 Overall Good UX

We will run live user tests to get a feel for what the users like and dislike about the design. See Usability Testing for more detail.

1.1.7 Sync Across Platforms

Server
Unit Tests
This feature depends upon the user_last_updated field for tasks in the database. Unit tests checking the last updated field were described in sections 1.0.3, 1.0.4, 1.0.5, and 1.1.2.

All of the update operations will require additional unit tests to ensure that the last updated field is checked before just updating the database. As merges may be required, this will be tested.

Any other tests of syncing across platforms will be done with integration testing.

Mobile
Dependent on the server.

Web
Dependent on the server.

1.1.8 OAuth with Google

Server
Unit Tests
After research, it was determined that this will be difficult to verify programmatically with automated tests. However, very simple integration tests will adequately ensure that authorization with Google is working correctly.

Bug Tracking

During the testing and development process, bugs will be encountered. When this occurs the developers will follow a specified process to track and eventually fix the bugs.

Bug Reporting Process

1. Create an issue on Github. This will allow us to keep a history of all bugs.
2. If the bug is something you will fix, assign it to yourself. Otherwise, assign it to the appropriate team lead who will then assign it to whoever will fix the bug. If it is a critical bug, notify the appropriate team lead outside of Github.
3. The assignee fixes the bug.
4. Before marking the issue resolved, one other person must test that the issue is indeed resolved.
5. If possible, an automated test case must be written for the purpose of regression testing.
6. Then the issue may be marked resolved.

Test Schedule

MVP (Version 1.0) Tests

1.0.1 Import from iCalendar feed

Server
23 March 2015 - 28 March 2015

Mobile
28 March 2015

Web
28 March 2015

1.0.2 Edit/Delete Courses

Server
30 March 2015 - 4 April 2015

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

1.0.3 Create Tasks from iCal Event

Server
6 April 2015 - 11 April 2015

Mobile
11 April 2015

Web
11 April 2015

1.0.4 Edit/Delete Existing Task

Server
13 April 2015 - 18 April 2015

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

1.0.5 Marking Tasks as Complete

Server
13 April 2015 - 18 April 2015

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

1.0.6 Task List View

Server
20 April 2015 - 25 April 2015

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

Version 1.1 Tests

1.1.1 Create a Course Without an iCalendar Feed

Server
27 April 2015 - 2 May 2015

Mobile
27 April 2015 - 2 May 2015

Web
27 April 2015 - 2 May 2015

1.1.2 User Creates a Custom Task

Server
4 May 2015 - 9 May 2015

Mobile

4 May 2015 - 9 May 2015

Web
4 May 2015 - 9 May 2015

1.1.2.a Name, Due Date, Course, Description

Server
4 May 2015 - 9 May 2015

Mobile
4 May 2015 - 9 May 2015

Web
4 May 2015 - 9 May 2015

1.1.3 Sort Tasks by Class or Date

Server
No tests are required.

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

1.1.4 Flag Task if Past Due

Server
No tests are required.

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

1.1.5 Completed Tasks Move to a Different Section

Server
No tests required.

Mobile
23 March 2015 - 28 March 2015

Web
23 March 2015 - 28 March 2015

1.1.6 Overall Good UX

Mobile
Constant

Web
Constant

1.1.7 Sync Across Platforms

Server
25 May 2015 - 6 June 2015 (includes integration testing)

1.1.8 OAuth with Google

Server
11 May 2015 - 23 May 2015 (includes integration testing)

Appendix

A-1. Devoir Functional Requirement

The Devoir functional requirements have been previously defined by our team in our requirements document titled Devoir Requirements Document.

A-2. System Usability Scale

Participants are asked to score the following 10 items with one of five responses that range from Strongly Agree to Strongly disagree:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.