Test Plan: Forms and Tasks - Fieldwire

Daniel Yang

Introduction

This test plan outlines the high-level strategy for testing the Forms and Tasks features. The goal is to create 2-3 automated tests for each of the features. The test framework should be designed to facilitate testing the two features and prepare for testing the entire Fieldwire product.

Scope

This plan covers

- implementing a test automation framework
- 2-3 automated tests for the Forms feature
- 2-3 automated tests for the Tasks feature

Assumptions

This document assumes the following:

- code style
    - the styling used in this project follow Prettier's formatting guidelines
- no device requirements
    - Tests run on GitHub Actions using a docker container with default settings for processor and memory availability
    - Local development is done on a Mac Mini
- Automated tests would be run in a CI (continuous integration) environment

Strategy

1. Investigate and review the features offered in the Tasks and Forms features
2. Document some test cases that could be created for the features
3. Implement an automation framework which can automate tests
4. Design the automation framework to be a scalable starting point for Fieldwire teams to continue with

5. Implement 2-3 automated tests
6. Implement tests in CI (GitHub Actions)

Out of Scope

- Usability Testing
- Integration Testing
- Performance Testing
- Regression Testing

Test Area Investigation

Forms

Forms is a feature of Fieldwire for creating and managing forms and their templates. Forms can be created from templates. Templates can be created from scratch or imported. The templates can include fields of various data types, custom status, and be published or in draft. Once a template is published, a form can be created from the template.
The E2E testing of this feature can be split into two categories. The two categories are forms and templates.

Tasks

Tasks acts as a project management feature and includes options for visualizing the tasks in a couple different ways: Kanban, Calendar, and Gantt chart. As part of the project management features, tasks can be prioritized and organized into groupings showing their current status. Checklists are a sub-feature of tasks. Each task can have a checklist of items. These items can be created as individual items, in bulk, or imported.

Test Cases Drafts

Each column contains a node, items in the next column are sub-nodes. {} indicates a list of different entry options to follow. sub-nodes should NOT be created for each of the entry options. [] indicates a list of different actions to follow, sub-nodes should be created for each of the action options.							
Tasks							
	new task						
		{task title}	valid string	string with spaces	string with symbols	non-utf8 characters	non-latin characters
		edit task title					
		unlink					
		[related tasks]	new task	existing task			

		[checklist]	new item	add checklist			
			{item name}	valid string	string with spaces	string with symbols	non-utf8 characters
			new checklist				
				copy checklist			
				paste checklist			
			[add existing checklist]	no existing checklists in account	import account checklist		
				manage existing checklists			
					search		
					delete a checklist		
					create new		
					import account checklist		
Forms							
	templates						
		new template					
			blank				
				[dated template]	TRUE	FALSE	
				{template name}	valid string	string with spaces	string with symbols
			upload pdf				
				[dated template]	TRUE	FALSE	
			pre-built templates				
				Daily report			
				inspection request			
				safety audit			
				time & material tag			
				timesheet			
		import template					
		select					
		filter					
		templates					

Decisions

I created test cases using an excel sheet and used rows/columns to annotate for myself the user flow. It is not a very easily digestible format, but is free. I also like how it provides me the ability to view where tests can loop through to prevent creating duplicate work.

I chose Playwright as the framework for test automation because I am familiar with it and its performance has been generally better than the main alternatives of Cypress and Selenium. It also provides a decent developer experience with its trace viewer and integrations into VS Code. With Microsoft support for Playwright, I can generally assume Playwright will work well with Github.

Once I identified a couple flows through the product, I chose to stop investigating the features available. Automated testing of features such as PDF export or printing would increase the complexity of this take home exercise and potentially prevent me from completing it in time.

I found the `data-e2e` attribute used in some places and decided to use a locator strategy based on them. I did not investigate accessibility locators, though I found some aria attributes. There were not enough `data-e2e` attributes for all of the elements and I ended up keeping my automated tests limited to areas where those elements existed, when possible.
For managing locators, I chose to use the page-object-library strategy documented in Playwright's documentation. It works well enough for a small project and can be scaled up to a large project.

I found logging in was necessary so created some test cases around those.
The tasks page requires time to update the task count in priority columns. I used Playwrights expect().toPass() to dynamically retry assertions until the page has updated.

Trade-offs

The page-object-library strategy creates a lot of boilerplate code that is not necessary for such a small amount of automated tests. Larger projects could benefit from them, but would likely need buy-in from the teams and coordination with developers to ensure front-end changes include the locators.

I chose not to focus on making the tests run in CI until the end of the project. I thought this might save me some time as I am confident in my ability to set up Playwright in CI. However, the tests did not immediately work which created some anxiety. There were some configuration issues I solved. I would set up the tests to run on commit and consistently push

to CI in future exercises.

Assertions are mostly created in the page-objects. Some were created in the test cases instead to save time and prevent from rewriting parts of the page-objects. Potentially, more classes could be created to contain those assertions.

I designed the automated test cases around the concept that each test should assert one thing. As an E2E test, this means many tests will likely overlap. Deleting a Form Template might require first creating one. That could, in theory, cover both test cases. However, I have found that for traceability, this presents a logistical issue as tests and teams scale up. The downside to this is that the more tests there are, the higher costs are for running these tests. Parallelization reduces some time, but the CI costs still increase.

Things that could have been done differently

The test clean up for some test cases uses api calls to delete the created form templates. The clean up should be done in the after step to ensure they are always run. This also keeps the tests focused on a single assertion. I didn't have time to refactor those out.

Playwright includes an option to have a special project run before others to generate login credentials. I believe I may have triggered an account lock-out with my testing in CI with a misconfigured password. Using this single run to generate credentials then copying the cookies and cache into other test runs would have prevented this.