CambiaAnswers

Alex Carson Bull

May 2018

1 Introduction

This file will go through my mindset behind the decisions that I made while answering the Cambia interview questions from Daniel Dinh.

2 Questions

2.1 Gherkin

From my reading about BDD the idea is to design a human readable file that shows how a user will flow through a program. While this program is more likely to be used in a back-end scenarios the concepts still apply. I tried to apply the ideas though and also gained a grasp of the best situation to use BDD. BDD is about defining the different smaller ways that a user can work through a product. By defining this you are setting the system up to function exactly as planned even with a large team working on it. If you define the contents of a feature ahead of time down the road you have a map from documentation right into how the code should function. This can then be used to keep backward compatibility as the software lives out its life. If you connect it with an automated test suite then you have the ability to rapidly test where the behaviors might be acting strangely and solve them before the code ends up back on a production branch. So for me, BDD and feature files are all about seeing how the user will interact with the product and making sure that the developers design to this specification. It is almost a universal design doccument that should be readable and usable by all parties during and beyond the development process. It lacks some detail because of this but gets the major ideas across.

2.2 Tools

1. Pros: Git is an amazing way to manage projects from several different perspectives. It allows multiple people to be working on functioning versions of a project at once and still keep the older versions. By branching it creates a workflow that can all build back into a central deployment branch.

Cons: Git can be a hard tool to learn and use. I first tried to pick up git on my own before going into school and it was a complete failure. In school most teachers would give quick tutorials and then assume students got everything. From my experience that is never the case. I still have a limited git skill set and I attribute that mainly to practicing on the GUI and slowly working into the command line.

2. Pros: Docker makes deployment of project seamless and easy. It builds containers that have the core modules that you need for your project. It also works on the kernel level so it is ultra fast. When you use this for multiple different employees you also get added benefits. All the employees are running their code in the exact same Docker environment so they won't have issues running like they are different machines.
   Cons: I have limited experience with Docker so most of this came from my time working on this project or research I did. I would say that Docker can be daunting at first, but I think it is more manageable than git. While I got up and running super fast I still am having trouble getting the output file out of the container. I also read about people having issues with different python images on the docker hub. This didn't effect me but could be an issue.

3. Choosing a language can depend on a lot of different things. In the work environment, it can be dictated by having people work on a more common language that the entire team knows or something that is good for the project. Languages often end up being heavily used for specific tasks(Python for science stuff, Java for enterprise applications, C for embedded, etc) and so you have teams built around knowledge sets. This increases collaboration and knowledge of the codebase in general. For this project, I picked Python because of the fact that I am learning new skills in other areas. I feel super confident in Python but have a limited knowledge of Docker and no knowledge of Cucumber Gherkin. I plan to spend more time on those areas and use python to save time.

2.3 Methodologies

1. I think a QA engineer has a unique role on any development team. They bridge the gap between the customer and the development team to ensure a seamless delivery. Today on large projects you often have people working on smaller chunks to build the big picture. Because of this, the end goal might not come together perfectly. A QA engineer should help the team by looking at the project and giving them the tools to make it come together. This requires ironing out the details of the project and giving the dev team the resources they need for a smooth transition into production.

2. From a BDD standpoint, I think you need to be going through the documentation and figuring out the workflow. The mindset behind this being that if you have the tests written out in simple human readable formats

then the team will have an easier time of putting it into code in a way that functions together. As always you should be working with the team to know where they want to go, what technologies they want to use and refreshing or learning about them. I think this should be a rough process though. You need to work for the situation.

3. In my opinion, automated testing has two major use cases. The first is when a system is critical and you need to make sure that it has been tested rigorously, that's not to say that manual tests can't be as rigorous. The other situation would be for simple tasks that it is more cost effective to spend a little more money upfront to develop a simple test. The counter to this is that some tests won't be automatable in an easy way. Humans have the skill to recognize patterns that machines have trouble finding. In these cases, it is worth it to set up a system that has a human check for faults.

4. To me, this problem breaks down into two sections. First, you are going to want to test as much of the code that has changed and the related code with it. If you changed something at a very high level that might only be used once this can be simple. If you changed some core features then you have to narrow in on the features that are a higher priority than other features. For example, if the team changed the front-end load balancer that connects with numerous different service then you would want to test the services that are called on the most often. There also might be a service that is rarely used but is mission critical. Depending on the project this is going to change. Creating a loose list of things to be tested would be a good idea in my mind then work off that.