I'm looking for a new job as a full-stack web developer, so thought it might be useful to me and my potential interviewers to share the answers I wrote recently to some questions I was asked in an interview. Happy reading!

What is important to you in a career?

I want to use my gifts to serve other people and provide for my family, and do the best job I can at that.

Have you ever worked remotely? If so, what challenges did you face and how did you overcome them?

Yes; nearly all of my 19 years of web development work has been done remotely. One challenge I faced was that a client asked for way more work than I could perform, and when I could not complete all of that work, he refused to pay for the work which I had actually completed! I had to let that client go. I learned a lot from that, but especially to communicate with a client early, often, and specifically about the client's needs and the project's progress. In my current job, we use Slack all day, occasional phone calls and screen sharing, Trello for issue tracking, and (rare!) lunches at a restaurant, and this regular communication helps us get things done. Most of the time I'm free to work without distraction, and we have open lines of communication to get help when we need it. For me, this arrangement has been ideal.

Describe the problem solving methodology you would use if asked to implement a new feature in an existing codebase.

Here's how it works in my current job. Typically I will get specific instructions from the product designer describing the new feature, sometimes with screenshots of how it should look. When something isn't described clearly enough or it doesn't quite make sense, I work with the designer to nail down the specifics so I'm sure I've got it right. If the feature is really large in scope, I document the feature's design first, starting with the intended use case, then the UI components & functionality, then the data model or database schema, then any new modules of code needed, then any general API surfaces that are needed. All documentation is versioned

with the code.

Then I move to implementing the code. I start by making a new feature branch using git flow. I add the feature branch's name to the Trello card. If I don't already know where the code needs to be modified, to find the right location at which to begin modifying the code, I often start from the user's first point of interaction with the app, so I inspect the HTML source and trace the execution and the data back through events, variables, HTTP requests, functions, back to the database, until I find the location where the new feature can be implemented. I often then use a debugger to step through the execution to be sure of its flow and to know exactly what data is present. In the places where we are able to use unit tests, I write a unit test first. Then I outline the code changes I think need to be made in comments, then I write the code (including docblocks for methods), run it, see if the unit tests pass and the user interaction works right, and debug as necessary.

After implementing the code, I briefly document the changes I made in a git commit message (formatted for conventional-changelog), include the Trello card's URL in the commit message, rebase the feature branch on the develop branch, squash the feature branch if appropriate, and merge it to the develop branch. After testing the new feature and getting approval from the product designer, I merge the changes from the develop to the master branch and push them to production. Then I record any useful notes in the related Trello card (noting the relevant git commit hash; I like how Github automates that part) and move it to the "Deployed" list. That way, if we discover later that the problem isn't really solved, we have a paper trail to follow to fix it better in the future. In a hobby project I used TravisCI to push changes to master & production only when their unit tests passed, to implement a CI/CD deployment pipeline.

How have the SOLID principles influenced your code?

Most of my code has not used complex class hierarchies, and most of my time has been spent writing code that depends on base classes provided in frameworks and libraries, rather than implementing the base classes myself. So I haven't focused much on the whole domain where SOLID principles matter the most. But I have learned aspects of SOLID principles and practiced them as I've matured as a developer. That learning started when I read about extreme programming on the C2 wiki and elsewhere in the early 2000's. While I care to use many best practices (e.g., I aim to write clean, well-organized, maintainable, well-documented and self-documenting code), I also remain a pragmatic programmer, rather than a purist.

Single responsibility principle: I have followed the single responsibility principle more since I learned to write functions so that they can be more easily unit tested (functions that do only one thing are easier to test), and because "don't repeat yourself" is good advice most of the time.

Open/closed principle: I've extended base classes plenty in Backbone, TurboGears, Django, and CodeIgniter, and then overwritten the default methods they provide. Ever since learning how Prototype, Mootools, and other early JavaScript libraries modified global prototypes and so made themselves incompatible with each other, I've avoided monkey-patching base classes--who knows what havoc monkey-patching them would cause somewhere else in the application!

Liskov substitution principle: When I've extended a base class and overwritten the default methods it provides, I've generally assumed my custom methods should return the same types which are intended to be returned by the default methods, because I get that changing the return value to be different than what other programmers expect is not a nice thing to do! It's best to make code as easy to understand as possible, and avoid things that will confuse the next developer who reads the code.

Interface segregation principle: Earlier in my career I saved old code that was not used in case I wanted it again. Now I delete it and rely on git to resurrect it if it's needed again. The reason is that unused code is literally useless. It wastes developers' time and attention. I haven't written interfaces. But this same concern would motivate me to avoid writing interfaces that require subclasses to implement unused methods, and would motivate me to use other structures like multiple inheritance, mixins/traits, or libraries to implement only the functionality that is needed.

Dependency inversion principle: I don't think I've intentionally tried to make concretions depend only on abstractions.

Do you have experience with TDD? When is it helpful? When is it not?

Yes.

Test driven development is helpful throughout the whole software development lifecycle. Writing tests first does a good job of connecting an implementation to its user story, design and contract. Test driven development can help keep code and the coder focused on implementing the features that are actually needed (avoiding feature bloat) and promote better code organization, modularization, and separation of concerns, promoting its readability and maintainability. Future users can consult unit tests as a form of documentation-by-example to be sure they understand how each method is intended to be used. TDD provides the developer an immediate feedback loop which helps catch bugs early, immediately after they are created, before deploying to production, and this gives developers, management, and clients more confidence that the software does what it's supposed to do. TDD makes it easier to refactor with confidence you're not breaking something. TDD can save a significant amount of time because it makes it less necessary for a human to manually test the same features over and over. TDD is especially helpful for implementing continuous deployment and delivery, which makes it possible to deploy new features to production faster than some competitors in the same market. Tests can help isolate the problem when applications unexpectedly fail in production after they had been working fine for some time in production beforehand. That kind of failure really does happen(!), and it's challenging to solve those problems without tests, logs, and other forms of instrumentation. After a bug is found, tests can provide assurance the bug is fixed. The more mission-critical a piece of code is to the business, the more it's worth testing to be sure it's still working correctly.

But not everything needs to be tested. Tests aren't needed for exploratory code, temporary prototypes, one-off scripts, or 3rd party libraries, and complete test coverage is less necessary for parts of the code that are not central to the business logic. In practice, I care more to write unit tests for controller methods than for the UI or views. Code that requires lots of stubs and mocks to test can make testing more trouble than it's worth. In the short term, the time needed to write tests must be balanced with the speed at which new features need to be completed. Over the long term, tests should be viewed as a way to reduce the time needed for maintenance (bug-fixing) on the software. The short-term and long-term needs of the business need to be weighed to know how much time should be given to writing tests. Some legacy code cannot be tested easily; newer web application frameworks tend to be set up to be more easily testable.

On the whole, test driven development is more helpful than not.

What new technology have you explored recently and what did you like / dislike about it?

Polymer

I've used versions 0.5, 1, 2, 3 and the most recent pre-release version to make some hobby apps. Web components' encapsulation and composability are awesome; they are what I always wanted out of ExtJS, jQuery, Backbone and other frontend libraries. Their encouragement to just "use the platform" when you can is excellent; it is the future that's becoming the present, because the platform implements the W3C standards. For example, JavaScript now provides tagged template literals, which are an effective native replacement for JSX. Polymer's team has done a great job providing a stable upgrade path between major versions. Polymer never became popular, and its library-specific 3rd party tooling support isn't strong because it's bleeding-edge technology, but those down sides haven't bothered me much because Polymer is mostly W3C standards, which are undeniably popular and have excellent tooling support, frontend frameworks are beginning to use web components under the hood, and the Polymer team provides very well-designed starter kit apps to show how to put together a deployable app using the best current frontend tools.

Angular

I've used AngularJS (version 1) for the past 2 years at work, and am comfortable with it, but find applications made with it unnecessarily complex. One plus is that Angular's tooling support is quite strong.

In order to get comfortable with the newest versions of Angular, I have made some tutorial applications with Angular 7, and appreciate its use of more recent JavaScript and TypeScript features like modules, which make dependency injection a bit easier, and its use of components for organizing an app's functionality, but the application architecture it requires still seems more complex and proprietary than necessary. React, Vue and Polymer are easier to understand and use, which seems to me to be a very important advantage, because it can speed up onboarding new developers, can facilitate faster iterations and can better prevent the accumulation of legacy code which is hard to upgrade to newer versions of its dependencies, so ends up breaking.

Docker

That's exactly the problem my current employer needs to solve in the next couple years. I maintain and develop several CodeIgniter 2 applications which require PHP 5.6. In order to run

PHP 5.6 on my Ubuntu 18.10 machine, I had previously used the ondrej/php PPA from deb.sury.org which permits installing several versions of PHP in parallel, but because PHP 5.6 is no longer supported (as of Jan. 1, 2019), in order to continue using it I had to hold back (not upgrade) some Ubuntu system packages, and eventually that broke a few other packages. The production servers' system packages will need to be upgraded to remain secure, and to upgrade the system packages, the apps need to be migrated to use PHP 7.x.

So to begin that transition, and prevent my local system from having broken packages, I've begun putting the apps in Docker containers. I like that Docker isolates PHP from my system packages! It will also let us upgrade each legacy PHP 5.6 application to a newer version of PHP individually as we have time, and until then, it will let us run the PHP 5.6 apps on upgraded and so secure servers, and make deployment even more stable than the automated deployment we already have. So I'm getting a Docker setup working to help my coworkers continue to maintain these apps after my contract ends.

This problem my current employer faces has shown me that companies should invest some time in upgrading their stack to newer software in order to prevent their software from becoming unsupported, insecure, broken, or incompatible with newer software, syntax, architectures, paradigms, etc. and the new efficiencies and so business value they can provide. Upgrading too often will slow down new feature development, but not upgrading often enough runs a real risk of the software ceasing to function as a whole.