I recently [tweeted a comment](https://twitter.com/joshin4colours/status/755869819497385984) about software testers knowing about what's going on the world of automation. It got a pretty warm reception, so I think I'll expand on my thoughts.

Whatever your role in testing is these days, your day-to-day job will probably be enhanced by using at least some of the following approaches. At a minimum I'd suggest knowing what these terms mean and an example of how they might be used in a software development shop.

**Using a Continuous Integration Service**

One the biggest changes over the past decade when it comes to automation in software development has been task automation. In the past, things like building a particular version of an application, creating documentation or updating the status of bug reports were done manually. Some teams even had dedicated individuals who were the "build person" who were responsible for initiating a build. Doing tasks like this manually (or with heavy ties to individual people or machines) was time consuming and created annoying bottlenecks such as the build person taking a personal day and blocking new builds from being completed.

Luckily, continuous integration (CI) tools came to the rescue by allowing tasks to be standardized and automated. CI services essentially schedule and execute tasks that a regular desktop computer can do and have these tasks execute on target machines other than itself. Going back to the build example, instead of having Bob being responsible for manually creating builds on his machine a CI service can be set up to choose a target machine and execute a build on that machine. Not only does Bob not need to be physically present at the build machine but builds can occur any time, either scheduled or in response to another action. For example, Alice the tester may want a build of an application based on the latest changes to see if a bug has been fixed, and can initiate that build herself. This not only frees up resources from doing repetitive tasks but also gives more control to teams over individual and team workflows. You can also chain CI tasks together to further streamline some tasks. Learning how a CI service works is also a great introduction to automation without a lot of emphasis on programming.

Some popular examples of CI services are [Jenkins](https://jenkins.io/index.html) (free, open source), [TravisCI](https://travis-ci.org) (cloud-based), and [Bamboo](https://www.atlassian.com/software/bamboo) (proprietary) but there are other ones as well. Even more low-tech is using a tool like cron or Windows Task Scheduler for use on a single machine to automate tasks. CI is indepensible for developing software beyond hobby programs, and it's one place where a tester can really add value.

**Using Modern Source Control**

I should first point out that I love source control. When writing code (or blogs!) it's too helpful a tool not to use. For a tester who codes, it's a no-brainer. Even if a tester doesn't code, using source control in a modern way can be a big benefit when testing software. If you're not familiar with source control, see an older post of mine on the basics.

What do I mean by modern? I mean using source control that 1) integrates with other tools like CI server or bug tracker and 2) allows for using good team workflow practices like trunk-based development. Good source control allows individuals like testers to analyze changes and dig deeper into what's happening in a software project. A tester with access to a source control history and some basic training can ask questions like "Which files in the application have had the most developers work on them?", "Which files have the most changes?", "Which changeset contained the code that caused the bug?" and so on. This information can be helpful for looking for patterns and underlying causes of some issues.

Integrating source control with CI services can be even more powerful. Issues in bug trackers can have their statuses updated based on changes made by developers. Testers can request certain requirements be automatically checked on incoming code such as passing automated tests or code styling requirements. Builds and deployments can be initiated by changes to code. There are many possibilities in this case when source control is used well, which is one of the underlying concepts behind continuous delivery.

The current hotness in source control is [git](http://git-scm.com), which is free and open-source with a healthy ecosystem around it. There are several other options such as [Subversion](https://subversion.apache.org/) and [Mercurial](https://www.mercurial-scm.org/) (open source) and Microsoft Team Foundation (proprietary, obviously).

**Telemetry and Monitoring**

This is a topic I'm not as familiar with but is definitely of interest to testers. Monitoring is an approach where hooks are placed into an app that sends information back to the software creator about how the software is being used. This could include which back-end/server API functions are being called and in what order, which parts of a UI are being used and at what frequency, and so on. The goal isn't to send specific user information back to the development team but more general information about what parts of an application are being used and how. This helps provide insights into what end users are *actually* doing and how they *actually* use the app. Alan Page is a tester at Microsoft that has briefly discussed some of the [cool things he's seen done](http://angryweasel.com/blog/intelligence-and-insight/) with telemetry and monitoring.

Similar to mining source control history, using monitoring can help provide deep insights into how users interact with an app, how the app is actually used and how certain features are received. You can find out answers to simple questions ("How many people logged in last week?") to more specific and insightful questions ("How did users change their habits when feature X was released?"). These are the kind of questions that help testers execute better testing strategies, and overall help teams make better choices for users.

For more information, check out the [AB Testing Podcast](http://www.angryweasel.com/ABTesting/) with Alan Page and Brent Jensen. For how a mainstream product uses telemetry, take a look at how [Mozilla uses telemetry with Firefox](https://wiki.mozilla.org/Telemetry).

**Using Selenium Well**

Last and certainly not least, the [Selenium WebDriver](http://www.seleniumhq.org/projects/webdriver/) is a tool pretty much any tester who works with web apps should be familiar with. At this point, the WebDriver is a [standard](https://www.w3.org/TR/webdriver/) tool for automating driving browser actions, similar to how a human user would interact with web apps in a browser. It has several language bindings, works with several mainstream browsers, and is a great example of an extendible API that can be build on top of by a developers. In short, it's a good piece of work.

When used smartly, the WebDriver allows testers and developers to automate user acceptance tests that can be placed in a continuous delivery workflow. The WebDriver can also be used to write automated tests that run locally to double-check changes didn't break critical features in unanticipated ways. There's even uses for the WebDriver that extend [beyond functional testing](https://applitools.com/web-app-testing/).

For testers who are interested in learning to code, the WebDriver can provide a [good introduction to programming](http://simplythetest.tumblr.com/post/110279905950/choosing-your-automated-test-language-the-better). Automating test scripts can be a comfortable way to get familiar with programming without diving deep into programming language waters. It provides enough of a structure to get started and still get some good testing work done.

With these concepts in mind, embrace test automation whatever your role is in software development.