JAVAWORLD

FEATURE

16 ways to improve your code after it's 'finished'

Just because you've squeezed out the bugs and delivered a working program doesn't mean you should relax

By Peter Wayner
Contributing Editor, InfoWorld
MAR 2, 2020 3:00 AM PST

You've pushed the code through all of the tests and they came back green. The continuous integration pipeline ran all of the way through. All of the check boxes on the feature list are checked. All of the Post-it notes have moved to the completed part of the wall. Phew.

It's tempting to call the code finished and head off on vacation. You've earned it. The team has earned it. Let the code do its thing for a bit. Isn't that the whole reason we wrote it? To throw it over the wall where it could just hum along and do its thing?

[InfoWorld's 2020 Technology of the Year Award winners: The best software development, cloud computing, data analytics, and machine learning products of the year.]

Alas, the days of being complacent and sitting still are over. Nothing is ever finished these days. Just because you've squeezed out the bugs and delivered a working program doesn't mean that you should relax. There are dozens of things you can still do to improve your code. Some are the mark of the good citizen who cleans up for the next team to come along. Some are opportunities for growth and capturing new markets. Some are the start of a new journey.

Here are 16 things to do when you come back from a bit of relaxation and recovery.

[Also on JavaWorld: 13 rules for developing secure Java apps.]

Lint

The tool called a lint or a linter is like a code review robot that enforces hundreds of semantic rules. Perhaps thousands of them. Some were written by obsessive scolds from the programming sect that counts blank characters and berates those who use too many or too few. Some were written by serious people who have flagged subtle semantic patterns that can lead to security flaws later. Your programming team has probably chosen a collection of linters and now is the time to run them.

Profile

Don Knuth once said, "Premature optimization is the root of all evil," because it's silly to spend time improving the parts of the code that only run occasionally. Now that you're finished coding, it's time to fire up a profiler and look for those hot spots. It's often the case that 10 percent of the code runs 90 percent of the time. Sometimes there are tight inner loops that absorb 99 percent of the cycles. If you can flag them now, a few tweaks can really pay off.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

Remove debugging tools

It's tempting to leave in the verbose logging options embedded in the production code just in case, but once the code is running it's a good time to clean out those tools and turn off the debugging options. The extra data clutters up the computer and may even threaten performance by filling up some disk drive. Leave the debugging out of the production servers.

Analyze with AI

Old programmers used basic regular expressions and statements to look for issues; modern programmers also have artificial intelligence tools. Amazon's CodeGuru, for instance, is said to "leverage machine learning models" to search out bad code. It's a fully automated process

built upon profiling and tight analysis.

Curate data

When you're building an application, it's easy to take databases and log files for granted. Now that you're done with the app, it's time to start optimizing the database for speed and stability. Add indices to the right columns to speed lookup. Add mirroring and timely backup to improve stability after power losses or a disk crash.

[DevSecOps: How to bring security into agile development and CI/CD]

Now is the time to start weighing the costs of storage versus the costs of a data loss. Just how valuable are the log files? How much do they cost to maintain? How much does a geographically diverse backup plan cost versus how likely is a catastrophic failure of a data center? These aren't easy questions to answer but once you understand the cost of backup you can decide on how much you want to gamble. It's like a paid trip to Las Vegas, only you're rolling the dice with your career and the jobs of everyone else around you.

Optimize data flow

Many applications can benefit from fast caches, either on the server or spread out across the internet with a content delivery network. Adding a distributed memory cache or integrating a CDN is one of the simplest way to boost the performance experienced by users.

Optimize data

Not all of the data needs to be as big as it is. Images are one of the simplest places to reduce size without trading off too much. Stylistic details like elaborate backgrounds can be replaced by CSS instructions for gradient fills that take a tiny fraction of the disk space and bandwidth. Photographers and artists often love to keep as much information and detail around in case they need it, storing images in RAW format. Tools like ImageOptim will strip away much of the

unnecessary detail below the user's perceptual threshold while also removing extra EXIF values that track extraneous information like the lens of the camera. The result is faster downloads and lower bandwidth bills.

Add an API

Many architects begin with a well-structured API for separating the front-end display code from the business logic underneath, but sometimes there's a good opportunity to expand the use of the code base by adding another door or window. API toolkits like Swagger make this relatively easy by delivering the parsing, the routing, and even the documentation. If you have some functions that are good, clean entry points to your current block of code, gluing them to a new API can enable new options for automation and integration.

Bundle into a library

Some code you write will have a second, third, or maybe a fourth life as a library that's incorporated into other projects. A good architect will anticipate some of these options and divide the code up into libraries at the beginning, but sometimes inspiration strikes afterwards. Refactoring the code into a library is a good start at giving new life to the work.

Document

Hah! Documentation is less important today than it used to be, but it is still helpful in the right doses. If you're writing well-structured code with clue-filled variable names and simple structure, the code won't need many local comments. But it is still helpful to sketch out the basic role for each section and perhaps indicate how the data flows through the code. It is also helpful to point out some of the potential problems with the code and illustrate how the code recovers from exceptions—if it does.

[Also on InfoWorld: How to improve CI/CD with shift-left testing.]

Split into microservices

More and more architects are taking their grand vision and smashing it into bits, knowing it's often easier to maintain several small applications than one big one. Developers can work on the different parts at the same time, coding and testing independently before doing one final integration test. Projects often grow over time, especially as you're adding extra features toward the end. Sometimes the advantages of splitting the work into smaller pieces become clearer as you near the finish line.

Containerize

More and more code lives out its life in a container image that specifies the new source files and also spells out just which libraries and other services are required. Crafting this configuration file can be pretty simple in some cases but there's plenty of opportunity to be clever. Some teams like to split the code into multiple containers in case some might be used separately. Generally each microservice lives in its own container, but there might be a reason for them to share. Some developers are even advocating more extreme approaches like creating separate containers for each document. There is plenty of room for debate.

Deploy to serverless

Now that serverless computing options are growing more common, it may be smart to extract the essential functions from your code and deploy them to a serverless platform like AWS Lambda or Azure Functions. The bills are computed by the call, so you'll pay nothing for fallow periods when there's no incoming traffic. If your code is already well constructed and doesn't require any local state, it is often easy to extract the business logic from your app and rewrap it in a simple function call used by a serverless system.

Deploy to mobile

Most good web apps are built tor mobile displays now, and they often work so well on smartphones that there is less and less demand to create a standalone app. Plus the web doesn't force the developer to jump over dozens of hurdles and reviews just to get listed by the App Store or Google Play. But sometimes there are good reasons to turn a web-based app into a native iPhone or Android app, and there are several good tools that make the transition simple by wrapping up the web server with an embedded version of a browser.

Purists will argue that running JavaScript in an embedded webpage isn't really native, and they're right that performance can lag for some apps, like intense games, but for many apps it's the simplest way to get something into the stores. There are other advantages too. Native apps can control data delivery more carefully by caching large blocks of the website locally. This can save bandwidth for both the developer and the mobile user, making the interaction a bit faster and the bandwidth charges a bit lower.

Move to web

In most cases, it's much more work to move in the other direction and rebundle your app as a website. Unless you used one of the toolkits designed for web coding, you'll largely be rewriting the native code written in Java, Swift, or Objective-C. Still, building for the web browser can offer you freedom from the tyranny of the app store reviews and also the opportunity to service desktop machines with the same code.

Keep going

Some clever programmer rebranded the idea of rewriting our code because, well, the word "rewriting" makes it sound like you made a mistake the first time. "Refactoring" has a better ring. Refactoring doesn't admit prior mistakes and that makes it easier on the ego. The process of improving the code, often in little jumps, is a good idea to start just after you "finish." Little improvements and fixes can be rolled into the code immediately.

Many teams are constantly refactoring, shipping, or deploying new versions daily or even hourly. On their own, these small changes seem immaterial, but over the weeks and months they add up to significant improvements. The iterations come so often that they start to blur the line between finishing your code and starting up again. It's just one continuous cycle of improvement and deployment.

This story, "16 ways to improve your code after it's 'finished'" was originally published by InfoWorld.

Peter Wayner is contributing editor at InfoWorld and the author of more than 16 books on diverse topics, including open source software, autonomous cars, privacy-enhanced computation, digital transactions, and steganography.

Follow