Demo Data as Code

**AUTOMATION
HELPS
COLLABORATION.**

THOMAS A. LIMONCELLI

Engineers are often asked to generate demo data for various reasons. It may seem like this one-time task can be done manually and forgotten. Automating the process, however, has many benefits, and supports the inevitable need for iteration, collaboration, and future updates. When data is treated as code, you can leverage techniques from modern software engineering practices.

Many years ago I was at a company that needed to produce a demo version of its software. The demo would essentially be the company's software preloaded with fictional data. Salespeople would follow a script that would walk the customer through the features of the product. The script involved finding various problems and resolving them with the ease that only this product could provide.

Marketing would create the script, and engineering would create a dataset that would support the story.

Using live customer data in the demo was not an option because that would be a privacy violation. Even so, no one customer dataset could support the entire demo script.

This project had many red flags. Engineers were expected to work on it "in their spare time." That misunderstands and devalues engineering work. When nontechnical managers don't understand something, they often assume it is easy to do and, thus, obviously shouldn't

take very long.

More worrisome was the fact that this “spare time” theory was supported by the incorrect assumption that the project was a one-time thing. That is, the data would be generated once and be perfect on the first try; the engineers could then wash their hands of it and return to their regularly scheduled work.

This assumption was intended to be a compliment to the engineers, but, “Oh, please, this will just take an afternoon!” is not a tenet of good project management.

I don’t know about you, but I’ve never produced something for marketing without being asked for at least one revision or adjustment. This is a creative collaboration between two groups of people. Any such project requires many iterations and experiments before the results are good or good enough.

Marketing believed that by keeping the requirements vague, it would be easier for the engineers to produce the perfect dataset on the first try. This is the opposite of reality. By doing this, marketing unknowingly requested a waterfall approach, thinking that a one-and-done approach would be less wasteful of the engineers’ time. The reality is that a big-bang, get-it-all-right-the-first-time approach always fails.

The primary engineer assigned to the project quickly spotted these red flags and realized that to make this project a success, he needed an approach that would allow for iteration now and provide the ability to efficiently update the project months later when version 2.0 of the software would necessitate an updated demo.

To fix this, the engineer created a system to generate

the demo data from other data. It would programmatically modify the data as needed. Thus, future updates could simply regenerate the data from scratch, with slightly different operations performed on the data.

The system he created was basically a tiny language for extracting and modifying data in a repeatable way. Some of the features included:

- ➔ The ability to import data from various sources.
- ➔ The ability to insert predefined (static) data examples.
- ➔ Functions to extract data from one database, with or without clipping or filtering.
- ➔ Synthesizing fake data by calling function f .
- ➔ Transforming data using function g .
- ➔ Various anonymization methods.

This is not so much a new language as it is a library of reusable functions. New features were added on demand, adding functions as needed.

Because the demo data was being generated this way, it was easy to regenerate and iterate. For example, the marketing manager would come to us and say, “More cowbell!” and we could add a statement such as `GenerateAndInject(cowbell)`. The next day we would be told, “The cowbell looks too blue. Can it be red instead?” and we would add code to turn it red. Rerun the code and we were ready to show the next iteration.

Anonymization is particularly difficult to get right on the first try. People are very bad at anonymizing data. Algorithms aren’t always that much better. There will be many attempts to get this right. Once it is deemed “good enough,” invariably the source data will change. Having the process automated is a blessing.

The data was generated with a “program” that looked like the one below:

```
# Salespeople need to be able to show “problem X”.
# We found this data in customer1’s dataset, but we
# only need the first 200 rows:
AnonymizeAndInject(“customer1.data”, 200)
# NB: Approval to use customer1’s data is in ticket #12345.
# NB: Anonymization technique signed-off in ticket #45678.

# The next thing sales will demonstrate is what it
# looks like when Problem X is fixed.
# Function X that generates data that looks that way.
# It bases this off dataset2.data, provided by marketing.
GenerateAndInject(X, “dataset2.data”)
# There is a requirement that at least one “problem Y”
# will be seen in the data. We hand-created that data.
Include(“problem-y.csv”)
```

Notice that the example code includes comments to record the provenance of the data and various approvals. We’ll be very glad these were recorded if there are ever questions, complaints, audits, or legal issues.

This was so much better than hand-editing the data.

This approach really paid off a few months later when it was time to update the demo. Version 2.0 of the software was about to ship, and the marketing managers wanted three changes. First, they wanted data that was more up to date. That was no problem. We added a function that

moved all dates in the data forward by three months, thus providing a fresher look. Next, the script now included a story arc to show off a new feature, and we needed to supply data to accomplish that. That was easy, too, as we could generate appropriate data and integrate it into the database. Lastly, the new demo needed to use the newest version of the software, which had a different database schema. The code was updated as appropriate.

Oh, and it still needed to do all the things the old demo did.

If the demo data had been hand-crafted, these changes would have been nearly impossible. We would have had to reproduce every single manual change and update. Who the heck could remember every little change?

Luckily, we didn't have to remember. The code told us every decision we had made. What about the time one data value was cut in half so that it displayed better? Nobody had to remember that. There was even a comment in the code explaining why we did it. The time we changed every data point labeled "Boise" to read "Paris"? Nobody had to remember that either. Heck, the `Makefile` encoded exactly how the raw customer data was extracted and cleaned.

We were able to make the requested changes easily. Even the change in database schema wasn't a big problem because the generator used the same library as the product. It just worked.

Yes, we did manually go over the sales script and make sure that we didn't break any of the stories told during the demo. We probably could have implemented unit tests to make sure we didn't break or lose them, but in this case manual testing was OK.

Creating the little language took longer than the initial “just an afternoon” estimate itself. It may have looked like a gratuitous delay to outsiders. There was pressure to “just get it done” and not invest in making a reusable framework. However, by resisting that pressure we were able to rapidly turn around change requests, deliver the final demo on time, and save time in the future.

Another benefit of this approach was that it distributed the work. Automation enables delegation. Small changes could be done by anyone; thus, the primary engineer was not a single point of failure for updates and revisions. Junior engineers were able to build experience by being involved.

I highly recommend this kind of technique any time you need to make a synthetic dataset. This is commonly needed for sales demos, developer test data, functional test data, load testing data, and many other situations.

The tools for making such a system are much better than they used to be. The project described here happened many years ago when the available tools were Perl, awk, and sed. Modern tools make this much easier. Python and Ruby make it easy to create little languages. R has many libraries specifically for importing, cleaning, and manipulating data. By storing the code and other source materials in a version-control system such as Git, you get the benefit of change history and collaboration through PRs (pull requests). Modern CI/CD (continuous integration/continuous delivery) systems can be used to provide data that is always fresh and relevant.

Ideally the demo data should be part of the release

cycle, not an afterthought. Feature requests would include the sales narrative and supporting sample data. The feature and the corresponding demo elements would be developed concurrently and delivered at the same time.

CONCLUSION

A casual request for a demo dataset may seem like a one-time thing that doesn't need to be automated, but the reality is that this is a collaborative process requiring multiple iterations and experimentation. There will undoubtedly be requests for revisions big and small,

the need to match changing software, and to support new and revised demo stories. All of this makes automating the process worthwhile. Modern scripting languages make it easy to create ad hoc functions that act like a little language. A repeatable process helps collaboration, enables delegation, and saves time now and in the future.

Related articles

Data Sketching

The approximate approach is often faster and more efficient.

Graham Cormode

<https://queue.acm.org/detail.cfm?id=3104030>

Automating Software Failure Reporting

We can only fix those bugs we know about.

Brendan Murphy

<https://queue.acm.org/detail.cfm?id=1036498>

Going with the Flow

Workflow systems can provide value beyond automating business processes.

Peter de Jong

<https://queue.acm.org/detail.cfm?id=1122686>

Acknowledgments

Thanks to George Reilly (Stripe) and the many anonymous reviewers for their helpful suggestions.

Thomas A. Limoncelli is the SRE manager at Stack Overflow Inc. in New York City. His books include *The Practice of System and Network Administration* (<http://the-sysadmin-book.com>), *The Practice of Cloud System Administration* (<http://the-cloud-book.com>), and *Time Management for System Administrators* (<http://shop.oreilly.com/product/9780596007836.do>). He blogs at *EverythingSysadmin.com* and tweets at *@YesThatTom*. He holds a B.A. in computer science from Drew University.

Copyright © 2019 held by owner/author. Publication rights licensed to ACM.