CRISCO

rena coen & ricky holtz

For project 2, our goals were straightforward: we wanted to maintain our simple interface, and figure out what it takes to make a good URL shortener that people would actually want to use. We wanted to use more powerful tools and employ some of the more sophisticated techniques that we had learned over the course of the semester, and ultimately, we wanted to create a strong URL shortening application that anticipated user needs, demonstrated technical skill, and - perhaps most importantly - had a decent sense of humor. This technical documentation goes over our process, tools, and considerations for the future of the Crisco URL Shortening-er.

MORE ROBUST DATABASE FEATURES & "LOG"/SERVER OUTPUT (2 FEATURES)

Problem: We wanted a more robust back end than what was previously implemented in Shelve, with more than two columns, and the ability to do SQL inserts, queries, and sorting (which are easier to read in code than working with a list of lists). For testing and demo purposes, we also wanted a way to track site interactions in real-time.

User Walk-Through: Nina visits Crisco to shorten a link to "Facebook." At first, she tries to use a tag already associated with a different site, but on her second try, she succeeds in finding a unique tag that's now associated with the Facebook domain. Later, somebody tries to enter the same Facebook/tag combo that Nina uses, the database recognizes the combination as 'valid' and gives the user a confirmation. As Nina is navigating the site, we are able to see the functions as they're being called and the state of the database as messages print to standard output on the server.

Technology Overview: We chose to implement SQLite in place of Shelve. We created a four column database containing *tag*, *url*, *popularity*/hit count, and a *random* flag. We didn't use a primary key, instead preventing duplicates programmatically. We wrote a number of functions to perform various database actions (checking whether matches exist, inserting new records, etc.), each printing an announcement that it was being called so that we could track the flow of the user through the site, and the calls to our code. Originally, we used variables concatenated into strings for SQL queries, but after a bit of reading about how to escape web-input text to protect against SQL injection, we settled on parameterized variables in the form, *db_cursor.execute("select tag from crisco where tag = ?", (str(in_tag),))*.

Challenges: We encountered some threading errors and locks, because simply closing a single cursor at the end of the app.py script wasn't sufficient. Instead, we needed to open and close new cursors for each function. This makes sense, because someone might take many actions on a website, and multiple people may be doing this at once, but it was confusing to investigate, particularly since the database code was working outside of a browser.

Alternatives: We considered trying to move forward with Shelve, but it seemed like a sub-optimal way to interact with a database with more than a couple of columns, so we looked for a SQL-style option. We considered SQLAlchemy but SQLite had all of the necessary functionality and was more familiar. For the "log" messaging, we considered sending the messages to an actual file rather than using print statements, but this didn't serve the purpose of our demo (we did try doing this, but couldn't get the nohup log to catch the printed database state). We decided that since this was an additional feature, we'd prioritize our demo over what we might want to use long term on a live website.

Future Improvements: For best practices' sake, a next step would be to put a primary key on tag & url. There are a lot of improvements that could be made for our output messaging. Even in its current demo manifestation, it would be nice to parameterize the print_db function such that it could take instructions for different column sorting if the programmer wanted to employ it. We didn't implement any real features around the short url hit count other than simply tracking it, but if we allowed site users to query this, certainly we'd want to enable other sorting options. Moving away from a debugging or demo manifestation of the formatted messaging, we would want to implement a true log file with fewer messages, and wouldn't want to print the database after every action taken, as it would fill the log quickly.

RANDOMIZED TAGS

Problem: Previously, our interface required both a unique URL and a unique tag in order to shorten a URL; this system worked, but it didn't seem to map to a user's expectations. We thought that random tags were a particularly important feature for user experience - sometimes, the user just wants to quickly shorten a URL and get on with their day! Now, Crisco's only requirement is a URL - it will automatically generate a shorter tag if the user doesn't provide one.

Example: Todd visits Crisco to shorten an image link he found on Google. It feels like it's hundreds of characters long, and he's just trying to send it to a friend on Google Chat. He plugs in the full URL of the image, hits 'submit' and gets a confirmation, alongside an automatically shortened URL. Later on, Kyle visits the page with the same image; Crisco recognizes the input URL, searches the database for random tags, and generates the same one for Kyle.

Technology: We used a simple python function and the SQLite database.

Challenges: Introducing any change to the workflow created a TON of different use cases for us to consider. Once we added the random tags, we had to start thinking about how to handle duplicates (both of URLs and tags), how random tags interfaced with user-generated tags, and a number of other issues that simply didn't occur initially.

Future: A smart recommendation system; sometimes, the shorter tag might not be as efficient as an existing tag for a site like Yelp. The tag 'yelp' makes more semantic sense and is shorter!

MOBILE RESPONSIVE CSS

Problem: Some of our text became unreadable on the smaller screen due to incompatible background/text colors, and the text also spilled off the screen because it was simply too large for mobile devices. We made some small but important modifications, implementing a single CSS file across all pages that responds both to device width and current screen width.

Example: User Tina visits Crisco Input on desktop, and enjoys it. The next day, she revisits the site on mobile; she notices that the site text has shrunk, elements in the DOM are now stacked, rather than blocked inline, and buttons are bigger and easier to press.

Technology: We chose to abandon the Twitter Bootstrap Framework and employ our own CSS; this gave us much lighter files to work with, and also gave us more control over each element's styling. This also helped remove some of the "Bootstrapped" feel from Crisco that so many other sites have, although we did keep the form elements and button. Our mobile responsiveness is based on a media query, and checks the width of the screen before applying the mobile declarations to the page.

Challenges: While removing Bootstrap seemed like the best solution at first, we had to go back and clean up our HTML classes, as well as our main CSS file. In the end, it helped, but initially, it might have actually been better to start from scratch, rather than trying to salvage the file from the old framework.

Alternatives: We've heard of both BoilerPlate and Foundation, but don't have firsthand experience with either.

Future: We have plenty of opportunities to improve our CSS, and implement stronger Javascript to give the page a 'mobile feel,' although our interface doesn't necessarily have enough touch interactions to warrant anything too robust. Perhaps an easter egg, instead?

FRONT-END OVERHAUL

Problem: Our initial site didn't make a lot of sense, and the file system reflected some of our initial confusion with Flask; it had a lot of duplicate files and redundant CSS. We cleaned up the front-end files, created a clearer 'brand' around the Crisco joke, and tried to capitalize on the whimsy of our 404 page to channel a sort of ironic Southern charm.

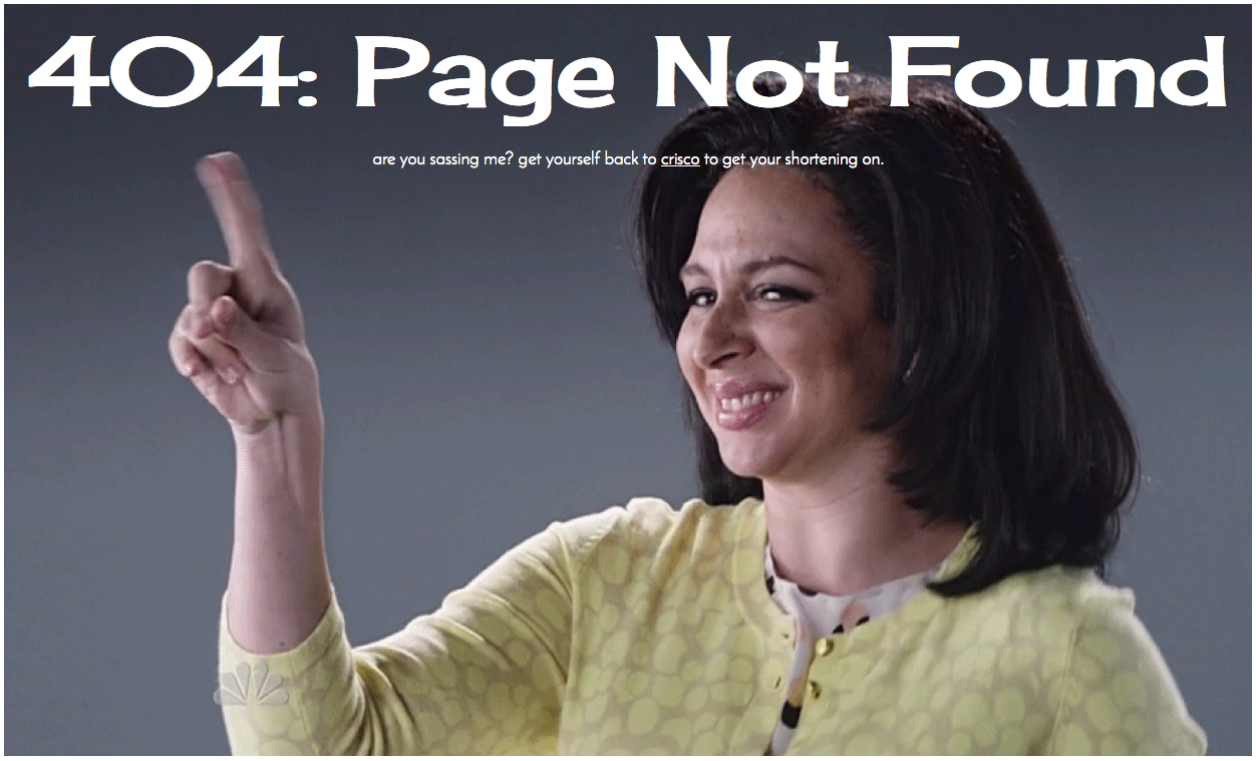
Example: Cam visits Crisco and notices that it's loading a few microseconds faster than usual. More importantly, however, the site has new backgrounds, new fonts, and new copy - oh, and suddenly they get the Crisco joke when they notice the pie in the background!

Technology: We spent minimal amounts of time in Photoshop, and frankly, an embarassing amount of time fighting with Illustrator. Most of our time was spent working on the Flask aspect of our program, including our static files and templates as we worked to make our links 'more flask-y.'

Challenges: This time, we developed Crisco locally, and the move to Harbinger wreaked havoc on our link structures. Everything seems to go relative to the root directory in Flask, and we couldn't find an elegant way to employ relative paths using Flask's link declaration for URLs and static/template files. At the last moment, we ended up needing to hardcode all of our links for Harbinger, which felt far less flask-y to us.

Alternatives: We've used some static site generators like Jekyll, but the interactivity and computation of Crisco made Flask seem like the best candidate for the job.

Future: We want to further explore template inheritance in Flask, as well as Flask's linking documentation, which was confusing, quite frankly. We like Flask's file structure, but we would have preferred to declare links that referred to that structure implicitly (e.g., {{url-for: 'static', filename=/imgs/404nope.gif'}}), not explicitly (e.g., src="/~ricky.holtz/static/imgs/404nope.gif").


404: Page Not Found
are you sassing me? get yourself back to crisco to get your shortening on.