Web Architecture 253, Fall 2014

Final Project Technical Report -- Blomo.link

John Semerdjian, Daniel Brenners, Wenqin Chen, Bryan Morgan

In this technical report, we outline the major features we chose to implement in our link shortening web application. Namely, we created user profiles, integrated it with the Wayback Machine API, designed it to be mobile ready, and implemented a PostgreSQL database. We have divided our report into technical descriptions of the front-end and back-end, while describing design decisions and challenges that we faced.

Front-end

Dynamic Page Rendering

One of the major challenges was the application is database driven. In order to do this we needed to incorporate a way to dynamically generate pages based off what was in the database. We looked into different solutions and decided the Jinja2 template engine would work well with the Flask server and our chosen theme.

Jinja2 is a template engine for the Python programming language. It provides Python-like expressions while ensuring that the templates are evaluated in a sandbox. It is a template language that can be used to generate any markup as well as sourcecode (http://en.wikipedia.org/wiki/Jinja_%28template_engine%29). These markups are called Jinja variables and are enclosed in {{ }}. They are placed in an html file at the location where the content enclosed is expected to appear. Jinja2 grabs these variables, which are defined in views.py in our case, and plugs them into the html files. When a page is rendered, Jinja2 automatically fills in these variable fields in the html.

Templates

The dynamic pages of our application are enabled by the Jinja2 template engine. The URL's of our pages are generated by using routes in views.py. This way, we can avoid hardcoding URL's and can update them easily. Also much of our other html contents are generated by Jinja2. We use it to grab variables defined in views.py, and plug them into the html files. When a GET or POST request is made, Jinja2 automatically fills in the variable fields in the template.

One of the challenges is how to render variables whose number of occurrence varies case-by-case. We wanted to avoid hardcoding divs and instead wanted to make the system dynamic. We ended up overcoming this challenge by using for loops with loop counters so that we could render only a set number of divs. Another challenge is we have multiple pages using similar styles. In order to avoid repeating the same styling code for each page, we decided to create two base html templates (base.html and initial-updated.html). We then used Jinja2 to extend these base templates to other html files.

We designed two pages for our user profile section, one for graphs and one for tables. We discussed two ways of implementing the routing to each page: create two separate routes in views.py or use one route in views.py to provide the data necessary for both graphs and tables. We chose the

later approach and used JQuery to show or hide information depending on which button on the left sidebar is clicked.

Mobile Ready

In order to make our application ready for mobile devices, we decided to use bootstrap with an existing theme. By using bootstrap and our theme we were able to leverage an existing system of media queries. The media queries allowed us to separate presentation from structure and subsequently dynamically represent the application to the user depending on their device. The breakdown for the media queries would essentially be, desktop, tablet and phone (with tablet and phone being quite similar). The media queries are able to detect the display size of the device accessing the site and then render the DOM in the most appropriate fashion.

Implementing mobile ready was relatively easy as we were building off an existing system. To integrate mobile ready into the templates we broke down the page components to the basic menu, which surrounds the content container. By doing this we were able to ensure that the presentation was consistent across different pages as well as on different devices. We used the existing class structure for the HTML elements to maintain the aesthetic and ensure that the framework was working properly. This was all relatively easy, but as we integrated bootstrap and our theme we began to realize there were some issues with the mobile ready implementation.

Once we had the framework implemented, we began testing and things seemed to be working well enough, but once we started testing edge cases we noticed some problems. One of the bigger issues we ran into was how the system was handling very long URLs. Setting up the tables to scale properly and not have overflow problems took some work. Eventually we used media queries to modify the behavior at different widths, which ensured that overflow was not a problem. As part of this, we made the design decision to only have the Wayback machine visible on larger displays (such as laptops/desktops or tablets in horizontal mode). The main reason for this was that while the Wayback machine is fun and interesting, it is not necessary. If a user is accessing the application from their phone, they are likely trying to accomplish something quickly and so providing extraneous features such as the Wayback machines is not helpful.

We ended up using bootstrap because this was the framework we discussed in class but we did also consider using uikit from Yootheme. This is a competing framework that allows significant modification before integration. One of the main problems for using uikit with this project though is the fact that it uses LESS, and would add an additional complexity in terms of development and deployment. In the end we felt that this complexity was not worth the benefit, at least for the scope of this project.

Dashboard

The goal of most URL shorteners is not simply to have a shortened URL for personal use, but also to share the link. For this reason, it is important, not only to shorten the URLs but also to keep tabs on the shortened URLs. The server is able to keep track of the number of times the shortened URL is used and this is exactly the type of information a potential user might need. With this in mind, we wanted to develop a way for users to understand what was going on with their shortened URLs in a way that made that information easily digestible. For this purpose we developed the user dashboard in the profile section.

A user simply needs to create their account in order to take advantage of the dashboard. Once the user has their account established, the system will keep track of shortened URLs and when they

visit their profile section, the first page has all of the basic information they would need. On the initial page, users are able to see three different pieces of information about the shortened links they have created. First there are up to 6 tiles, representing the 6 most recent links they have shortened, with daily breakdowns of the number of times users have used the shortened URL. The goal of these tiles is to allow quick access to more detailed usage information of the links that are most important. Below these tiles, there is a bar graph that shows total clicks for the 8 most recent links. This gives the user a general overview of how their recent links are doing. Finally on the right hand side there are shortcuts to the 13 most recent links and when they were created. This allows users to quickly access their more recent links, without having to go look for them. If the user still does not find what they are looking for on the initial page, then they are able to access a table of all their links by clicking on the button on the left.

In order to create the visualizations, we used the graphing libraries that were included with the theme we chose. We did this for two reasons: we knew they were already well integrated with this theme and the theme itself came with a substantial selection of different graphing libraries. From the selection of graphing libraries included, we ended up using Sparkline.JS for the tiles. Sparkline uses jQuery to create inline graphs on the client side, which meant that we were able to simply pass information from the backend to the client and have Sparkline dynamically generate the graphs we needed. The backend system was passing a large tuple that contained information about each link for the specific user and using Jinja2, we were able to parse out the relevant data for the graphs and place it within the HTML that created the graphs. This is the same pattern we followed for implementing the rest of the dashboard, using the data passed by the backend, then iterating through and parsing out the relevant information.

Wayback Machine

We provided users with a link to their long url from a decade ago using the Wayback Machine. We thought this was a fun way of reminding users how much the internet has changed. This feature is only available to registered users. When the user logs in to the profile section and navigates to their table of links, they will find a Wayback Machine link for each of the long URLs. If the site did not exist 10 years ago, users would be redirected to the closest snapshot the Wayback Machine has.

The Wayback Machine is a tool that was created by the Internet Archive, and stores 9 petabytes of information, snapshots of the web. These snapshots are available to the general public via their web app (https://archive.org/web/). In addition to their web app, the Internet Archive also has created an API for the Wayback Machine, though there are significant limitations to it. The API itself only has one GET methods and can only accept two options, timestamp and callback. The API returns a JSON object that only has the following information:

```
1. {
2.   "archived_snapshots": {
3.     "closest": {
4.       "available": true,
5.       "url": "http://web.archive.org/web/20060101064348/http://www.example.com:80/",
6.       "timestamp": "20060101064348",
7.       "status": "200"
8.     }
9.   }
```

```
10. }
```

This information is not very useful and will only return the closest snapshot. The URL returned in the JSON object is functionally the same as simply using the web application. We decided to simply use the Wayback Machine web application instead of trying to make calls to the API as this was a more direct approach.

We initially wanted to incorporate the Wayback Machine in an iframe on our site. We were able to pull into the web app via an iframe and were modify the view so that the entire page fit proportionally within the iframe. When viewed on a desktop, the Wayback Machine experience was suboptimal but still functional; on a mobile device it was simply impossible. We scraped the iframe and decided to create a link to a view of the Wayback Machine. While not the most elegant solution, it ensured that the feature was accessible on tablets and desktops.

Another option would have been to create another view that would display the Wayback Machine link in a large iframe, but nestled within our applications templating. This would mitigate some of the problems surrounding mobile ready and make the integration seamless. There would be some difficulties, even with this solution, as iOS devices don't honor overflow.

Back End

Views

All GET and POST requests are resolved by the routes defined in the views.py file. These routes include our root page ("/"), registration ("/registration"), login ("/login"), logout ("/logout"), shortened url ("/s" and ("/s/<url>"), user ("/user"), and profile ("/user/<username>"). The associated Jinja templates, forms, request methods, and login/logout process are contained here.

The primary task of associating a short and long URL occurs at the "/s" route. We initialize a ShortenFrom form object that will contain the data passed along from the POST request. We create some variables to store the necessary information and determine whether the form is valid and if the user is logged in. We also check if the short URL is already taken and display a message back to the user to select again. If all conditions are met, we create new Link object and add it to our database. The user is then taken to the shorts.html page to view their new shortened URL. A similar process takes place during user registration, clicking on shortened URLs, logging in, and viewing a profile page: authentication is determined, a call to the database is made, and a page is rendered.

The most important step in this process was keeping track of all the possible outcomes for a user to experience on any given page. We had to determine whether we'd allow POST or GET requests, what type of data had to passed through, whether users needed to be authenticated or not, and many other considerations. Writing clean yet explicit code helped keep this process organized and logical.

Login/Logout/Authentication

The Flask-Login module helped us manage user sessions and was simple to use. We created a user_loader callback which would reload the user from the ID stored in the session. This process authenticates the user but requires another function, login_user, in order to actually log them in. The

logout_user function, called in the “/logout” route, does the opposite. Pages where users must be logged in to view -- the user’s profile page -- are defined by the @login_required decorator. Flask provides g variable for storing data during the life of a request. Logged in users are stored in this global variable; if the user is not logged in, calling g.user will return None.

Forms

We developed three from classes for our web app and did all form validation on the back end. The first form is for user logins and includes fields for username, password, and remember_me. The username and password fields are required. The second form is for the long and short url form fields, which are also both required. The long url field must also be properly formatted URL, which should include both the scheme (“http://”) and domain (“www”). Long urls must also be less than 10,000 characters. The short url field only allows for alphanumeric characters; allowing users to enter a forward slash in their short url would have unintended consequences. Lastly, we created a form class for user registration. This form includes three fields for username, password, and email. All three fields are required and also have custom validators. The username field has a minimum and maximum length requirement, password a minimum length requirement, and the email field must also be a valid email address.

The form classes we created take advantage of the Flask extension for WTForms. This module allowed us to quickly develop validators and import them into views.py. For example, the registration route in views.py first creates a RegistrationForm object with the request.form as a variable. When a POST request is made we call the validate_on_submit() method which checks if all the form validation conditions are met. If they aren’t met, we flash a message to the user and redirect them back to the registration page. On the templating side, we created variables that are passed to the back end though POST requests. When form conditions aren’t met, we send back data to the client and give feedback to the user.

Passwords

To manage username and passwords safely and effectively, we hashed the passwords with bcrypt. While passwords are displayed in our application's login page as plain text, they are not stored as plain text. Passwords are salted and hashed using the Flask-Bcrypt module. Bcrypt offers a safer solution to hashing than SHA1 and MD5 because of its de-optimized design that, while not as effective for hash tables, increases the time it takes to possibly de-hash sensitive information.

Database

We implemented a PostgreSQL database for our web application. We initially chose a MySQL database, but switched to PostgreSQL since Heroku, our hosting service, already provided a free-tier PostgreSQL database. Luckily, the switch from MySQL to PostgreSQL proved to be trivial since we had used SQLAlchemy, an object relational mapper (ORM), to manage our database.

Object relational mappers allowed us to interact with our database as a collection of objects within Python, rather than interacting with tables using SQL. By telling SQLAlchemy to translate into PostgreSQL instead of MySQL, we were able to avoid rewriting much of our existing code. The URI for the database is contained in the db_create.py file, which relays information about the database we are using (MySQL or PostgreSQL) to SQLAlchemy and then creates the database.

The data we chose to store from the user had to account for all user information required for logging in and keeping track of the URLs the user shortened. To do this, we created three tables in our database: User, Links, and Clicks. In our ORM, these were represented as classes, or models, that would define the columns of each table. The User table would account for login information and contain the username, password, and email. The Links table would contain each link that the user created and have columns for short URL, long URL, and date created. The Clicks table would account for each time the shortened URL was clicked on and contain short URL and the timestamp for each click. Each table also had an ID column that served as a table's primary key. Foreign keys were also established in order to link tables together. This linking would allow us to join a single user to all of the links that they created, and link a single short URL to all of the clicks that are specific to that URL. This linking allowed us to describe each URL per user, as described in the users profile page. By linking the URL to its clicks, we were able to count the clicks per URL in order to display the analytics for the URL. This is seen in the models.py file.

The views.py file contains queries and logic that are performed once the user navigates to a certain resource. For instance, the shorts() method is performed once a user creates a new shortened URL. This method checks to see if the Link has already been shortened by querying the Link table and filtering by that URL. Unfortunately, the difficulties of SQLAlchemy combined with a few unforeseen complications that had arisen from using Heroku and PostgreSQL forced us to use a less than optimal approach to query the databases for the analytics visualization. Each time a user visits his or her profile, several operations occur that involve a combination of querying with the ORM and Python.

First, a list of links established by querying the database and using a list comprehension to go through the query and select each unique URL. For each unique URL in the list, a separate query investigates the number of clicks, and appends that number to a list. The order of the list of total clicks matches the order of the list of URLs. To get the number of clicks for each day, we perform a similar routine, but instead we filter the list of total clicks further by the timestamp. A list is created iteratively by filtering it seven different times, once for each day of the past week. We now can piece together these lists and create a variable for each user that contains his or her long URLS, shortened URLS, total clicks per link, a list of weekly counts per URL, and dates that the URLs were created. By returning this so called "master list" to the user profile template, the HTML can render these values in text and with a visualization.

Future Work and Improvements

Future improvements to our site apply to both the front and back-end. Front-end improvements include password masking, dynamic y-axes for our bar plot and line charts, and y-axis starting from zero for line charts, and also adding a link shortener directly on the user profile page.

On the back-end, we can improve the how user-specific click data is stored by caching results or by aggregating data on a less frequent basis (e.g. nightly ETL job). Given the scope of our project, we decided not provide account verification through email, though we could consider adding later. In addition, providing another field on the user registration page to confirm the typed password would also have been a good idea.

In the future there are a number of things that we would like to improve upon for the frontend. With the new technology being released for the modern web and with the increased "appification" of web offerings it seems that we are moving away from native applications and more towards web apps. To keep up with this change it would be good for us to go through and changing some of the media queries to make them more granular, and able to detect specific devices. This

would allow us to push the boundaries of what we can present the user with and really tailor the experience to the device, giving it more of a native application feel.

Additionally we could provide the user with greater granularity over how people are using the shortened URLs. This would be helpful for users focused on their social media impact. Currently we are only tracking clicks per day, but it would also be helpful to understand what time of day people are using the shortened URLs, or where the person using the shortened URL is located. This type of information could all be displayed on the dashboard and would make the tool much more powerful for users.