{Christina}

Hello everyone! I’m Christina from GitGoing, and I’m joined today by Michael, Travis, and Sarah. When last we left off, GitGoing was in-progress with several functionality requirements still needed. We talked about certain features that still needed implementation, such as authentication on log-in and registration, inline commenting, and messaging regarding reviews. We also needed a more robust database API that securely sent and retrieved information. Well today, we’re happy to present that many of these planned features are now live, and available for use.

{so this will address what we talked about last quarter}

//maybe brief images of the code that makes these things work. But will be displayed later so idk

We’ve talked about it before, so we won’t go to in-depth, but our log-in and registration systems are all fully authenticated by Congito from AWS

// Maybe 1/3 paragraph length

Our projects page displays file contents after securely retrieving from a DB

//maybe images of the route for the get(review), and images of the code on react.

Our projects have in-line commenting

//same as above

And our DB api is a robust secure system paired with the cognito-express AUTH library

//images of some of the routes and the auth set-up

And here’s Travis to talk about some of the other neat things we’ve done

-------------

{Travis}

Thanks Christina! So, in addition to all of these awesome goals we’ve achieved, we also took things quite a bit further in-regards to making the site, y’know, usable.

As y’all will be shown in our demo, once a project has been created users can be invited to collaborate on the project through our UI system on each individual project page. After a user logs in, they are redirected to the main site’s homepage, where any notifications they may have received are currently being displayed. These notifications display details about any invites they may have been sent, and give users the option to accept, or decline as they prefer. It’s also important that notifications are informative, so they include details regarding which project or file they’ve been invited to work, as well as the date/time the invite was issued. If a user declines an invite, it is removed from their notification queue. If a user accepts instead, they are then able to view the project, invite other collaborators, delete the project, or make comments on the file. They also have the option of making local changes and uploading a new file! After a user is brought on as a collaborator, or a file has changed in an existing project, you may want to alert others on the project that a review is required, so users can also invite others to review any changes. Users are given a similar notification as before, and if they accept the invite they are automatically redirected to the specific file so they can start reviewing! These redirects and page URLs are made possible through React’s dynamic routing abilities it offers developers

//code snippets

Now, all of this would be worthless without access, and we are proud to announce that Git Going is 100% live, available for use in its current Alpha state. Through multiple AWS services, the site is reachable at our specific domain name, and it processes requests as any normal webpage might, though perhaps more securely than some others. The gist of it is that we have a domain name registered with Route 53, and a certificate, issued by AWS, attached to the domain name that provides SSL/TLS (secure sockets layer/transport layer security) transmission. The build of the project sits inside an AWS simple storage service bucket (AKA S3 bucket). A bucket is simply a container for objects stored in S3, and its meta-data. Basically, the bucket “serves” specific files on received requests. Then, a global content delivery network, CloudFront, services requests by catching traffic heading to our specific domain name, converting that request from HTTP to HTTPS if its not already, and pulling the specific files from our S3 bucket. It then sends those files to an edge location, which is just a geographically close-to-requester data centers. The requester receives the files from the edge location, which allows for incredibly fast, and secure requests to our site! In keeping with this, here’s a brief example of our Node.js app security working with the private/public keypair that AWS issues. The API receives the public key in a header, then makes a call to the private key on the Node.js side using the cognito-express library!

-----------

So that’s the front-end. Now here’s Michael to talk about the back-end.

{Michael}

Thanks Travis. Hey everyone. So, at the end of the last quarter I had this flask app which could receive a request with a filename and return a diff between file versions, and I could do this on localhost for my own machine, but that was pretty much it. So obviously I needed to figure out a hosted backend solution and that was one of my biggest goals for the quarter.

There were two solutions I mostly considered, and one was AWS Lambda with Amazon Elastic File System. AWS Lambda allows for serverless execution of code (such as the flask app or our nodejs app), which can mount Amazon’s Elastic File System where the files would be handled to generate diffs. The other solution was to host our flask and nodejs apps on a server, and this is the route I ended up going with, because I think it gives us more flexibility in how we deploy our services. The server could be a cloud hosted virtual machine, it could be your home PC, it could be a raspberry pi, or a powerful superserver. I also thought it would be easier to go from this implementation to serverless code execution, since our flask or nodejs code would largely be the same.

So, with all that in mind, I got an Amazon EC2 instance running, which is just a cloud hosted virtual machine, and started doing some Vim coding through SSH which made me feel like I was doing something cool. This gives me the file system I need for getting diffs, as well as the host I need for the flask and nodejs apps. But now I needed to figure out how to get it so these apps could actually communicate, and this lead me down a road of learning about HTTPS and SSL (which is actually TLS now, but lots of people still call it SSL which is confusing, but sure). But basically, it’s an encryption protocol to provide security to HTTPS (the S in HTTPS is for Secure). To process https requests, you need a valid TLS certificate. You can use self-signed certificates for development, but as soon as you try to use this in a production environment, you’re going to learn that your browser will warn you, and https requests will be blocked. You’ll need a certificate from a Certificate Authority for production. These often cost, but luckily LetsEncrypt is a nonprofit Certificate Authority which provides TLS certificates for free. Unfortunately, their certificates expire after only 3 months, but you could at least set them up to auto-renew (they suggest every 60 days).

Once I understood all this, I was able to get both the nodejs app and an apache server for the flask app hosted and communicating with our hosted frontend with HTTPS requests. As you can see on the right, both applications are now using those TLS certificate files generated by Letsencrypt, and now our backend applications are live and hosted, and able to securely communicate with other applications across the web, which was an exciting goal for me this quarter. One small side note: From what I understand, NGINX would be a better solution over apache for the flask app so I’m looking into that, but I’m otherwise pretty happy with how things turned out.

But with that, now Sarah is going to show us a demo of how all these things currently work together.

{Sarah}

Thank you Michael!

{demo}

//flow of all working functionality

Conclusion: still to come, feelgood, everybody clapped