Design/Architecture Decisions:

We implemented a Discord bot intended to ease and automate much of the responsibilities of moderators and regular members of a Discord server. As such, looking into what actions are frequently performed and would benefit users most if automated helped us make decisions on what commands to implement.

Our original design had some fundamental commands that we felt were essential to a Discord bot:

- Ban
- Kick
- Temporary Mutes
- Message Filter
- Mass Deletion of Messages
- Polls
- Afk status
- Self assignment of roles

We felt that these commands were the core structure of any successful Discord bot.
While implementing these commands, we made sure to isolate each one in its own sandbox, in such a way that developing or testing one feature would have no impact on the others. By keeping them all in their separate modules, we were able to minimize regression by ensuring independence. We also made sure that once a command feature was done and merged into master, we documented the feature and its usage format on our README. This ensured that all progress was documented and could be easily tested by any member of the team. We kept the "src" and "team" folders separated to ensure that our source code remained distinct from our team folder, which focused on documentation of stand-ups, retros, etc.

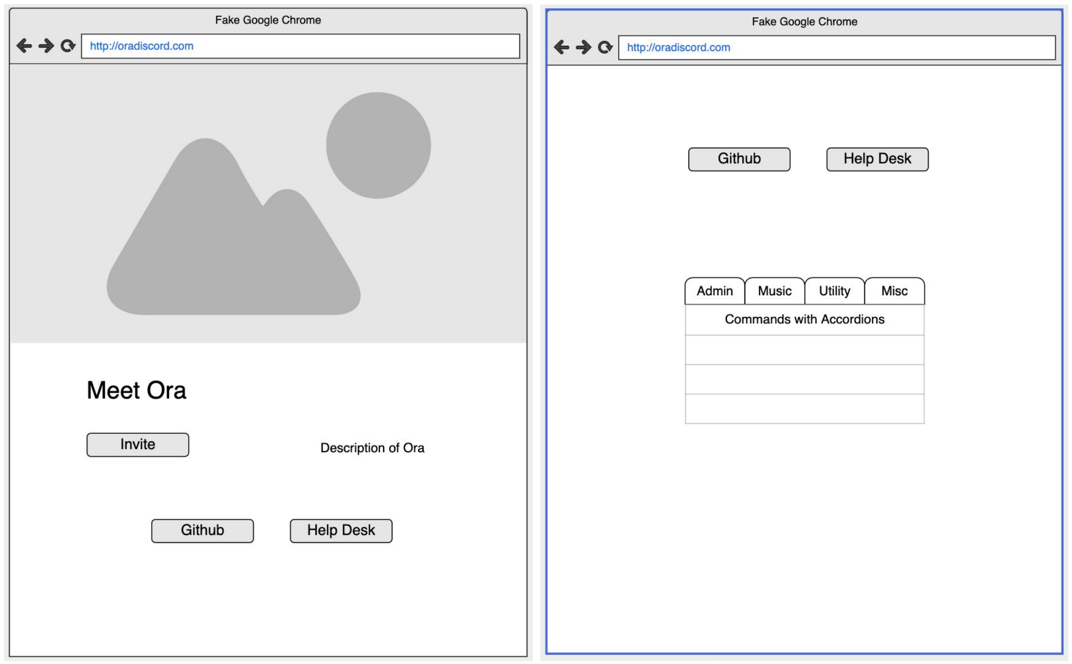
We chose Javascript and Node to be our foundational tech stack after much deliberation. Although few of us had extensive experience in any of these languages, we felt it was most fitting for a web application-related bot. We used this as a means of learning and using a new language. Though there was definitely a bit of a learning curve, as detailed in the Implementation Details section, we feel it was a good decision due to various useful packages (namely Discord.js, among others), the ease of integrating these packages with npm, and how much we learned about the foundational language of the browser/web.

We soon realized that Discord bots require a centralized location for documentation. Discord bots also require server owners or moderators to invite them to join a server. This meant users would need to access a website to be able to integrate and manipulate the bot within their own Discord server. We settled on React being our foundational tech stack. Some of our team had prior experience working with this framework and we felt like there would not be a steep learning curve. During the design of our web application, we realized we wanted a customizable CSS framework that would allow us to develop and iterate quickly. This led us to make use of

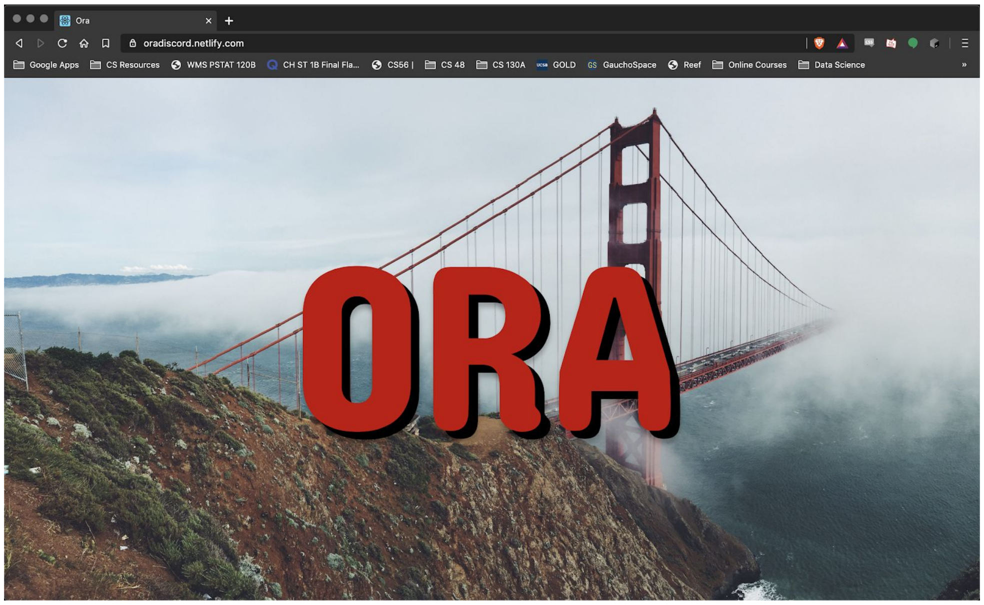
Tailwind. Tailwind allowed us to make use of descriptive utility classes rather than creating custom css classes. Furthermore, Tailwind allows developers to create custom components from the utility classes. This allowed us to quickly and efficiently build a responsive website that provided new users with documentation of the commands as well as an invite link.

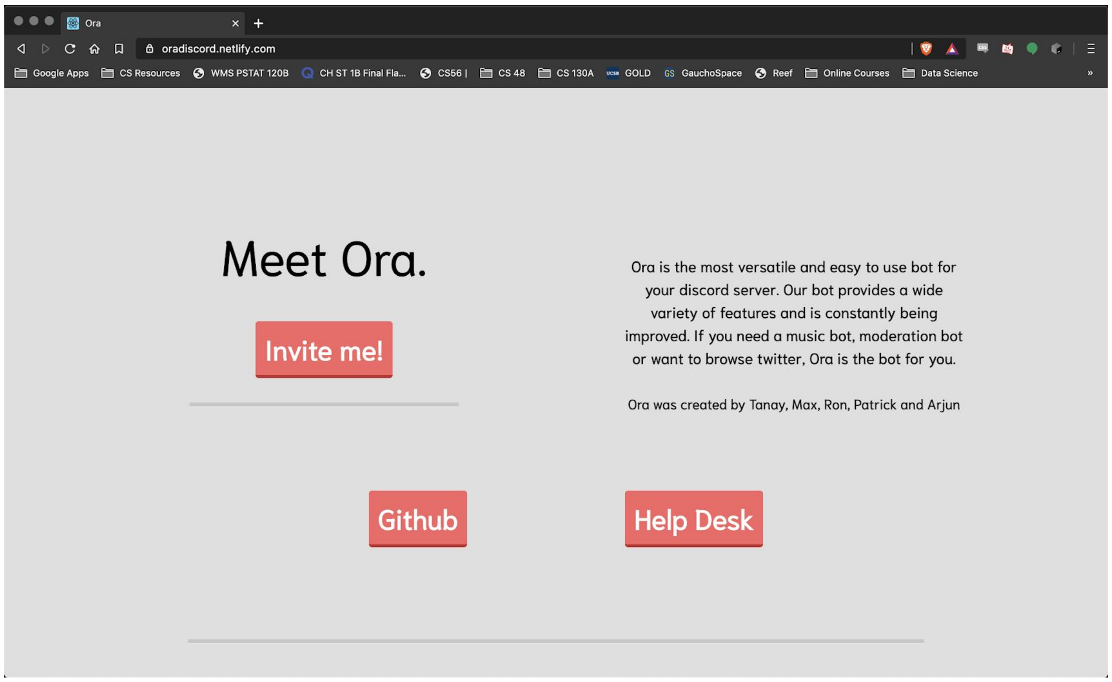
Stages of Design:

Initial Sketch:



We initially decided on a single page web application that provides users with documentation for the robot. They would also be able to access an invite link, our Github repository as well as a help desk.





Meet Ora.

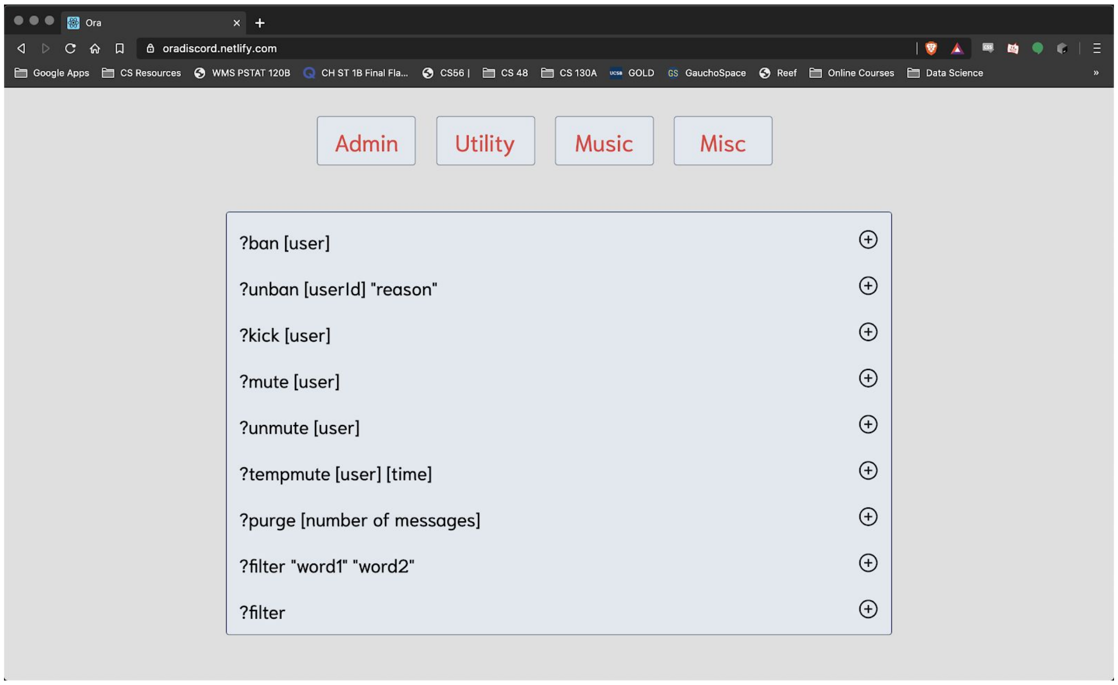
Invite me!

Ora is the most versatile and easy to use bot for your discord server. Our bot provides a wide variety of features and is constantly being improved. If you need a music bot, moderation bot or want to browse twitter, Ora is the bot for you.

Ora was created by Tanay, Max, Ron, Patrick and Arjun

Github

Help Desk



?ban [user]

?unban [userId] "reason"

?kick [user]

?mute [user]

?unmute [user]

?tempmute [user] [time]

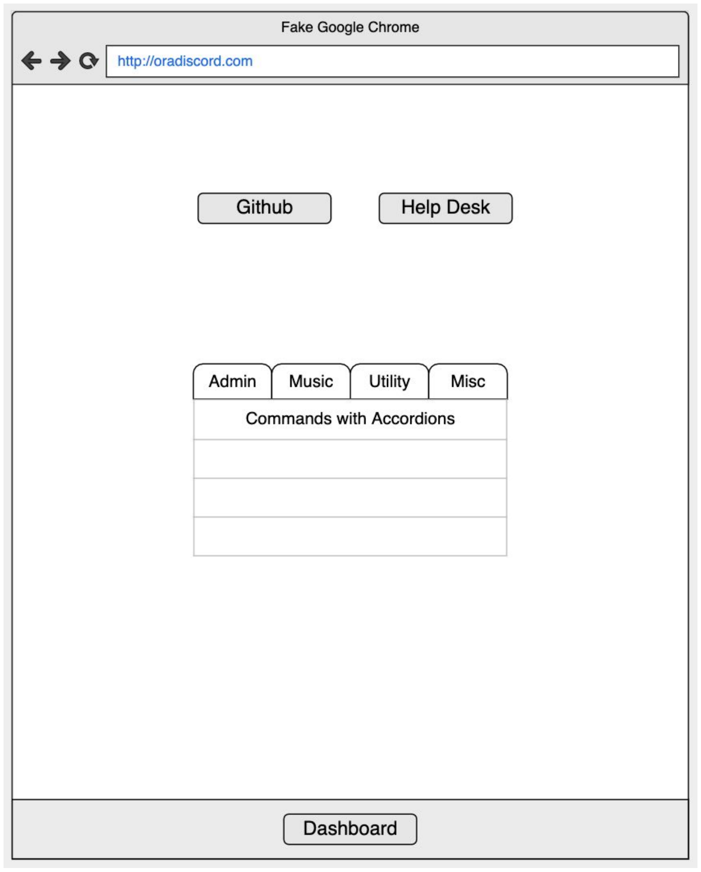
?purge [number of messages]

?filter "word1" "word2"

?filter

Soon we realized, many users would prefer to be able to customize Ora to suit their needs. This could include changing the syntax of the command, creating custom commands and so much more. As a result we began to look into creating a user dashboard.

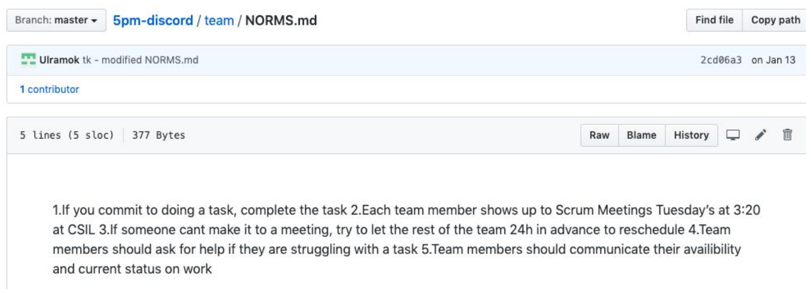
Initial Sketch:



We decided to create a website with a sticky footer that provided a button the user dashboard. The user would then be able to log in with their Discord account via Oauth2. This would allow us to view the servers they could manage and allow them to modify their server settings. We would then store the server data in a mongoDB atlas instance. However, we ran into issues during implementation and as a result, this did not make it to the released version. This feature is something that we plan to work on in future iterations of Ora.

Programming documentation:

In our first meeting, we wrote about ourselves and decided on team norms that all members would be accountable for:


Branch: master 5pm-discord / team / NORMS.md Find file Copy path

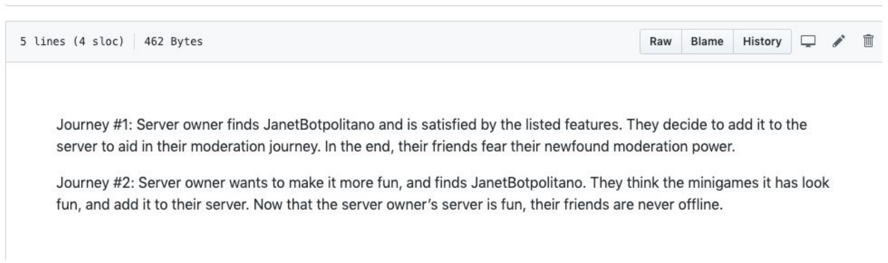
Ulramok tk - modified NORMS.md 2cd06a3 on Jan 13

1 contributor

5 lines (5 sloc) | 377 Bytes Raw Blame History

1.If you commit to doing a task, complete the task 2.Each team member shows up to Scrum Meetings Tuesday's at 3:20 at CSIL 3.If someone cant make it to a meeting, try to let the rest of the team 24h in advance to reschedule 4.Team members should ask for help if they are struggling with a task 5.Team members should communicate their availibility and current status on work

These norms helped us establish dependable practices that we could expect from each other. We also wrote user journeys to help us look at our product from a consumer's perspective rather than a development perspective:

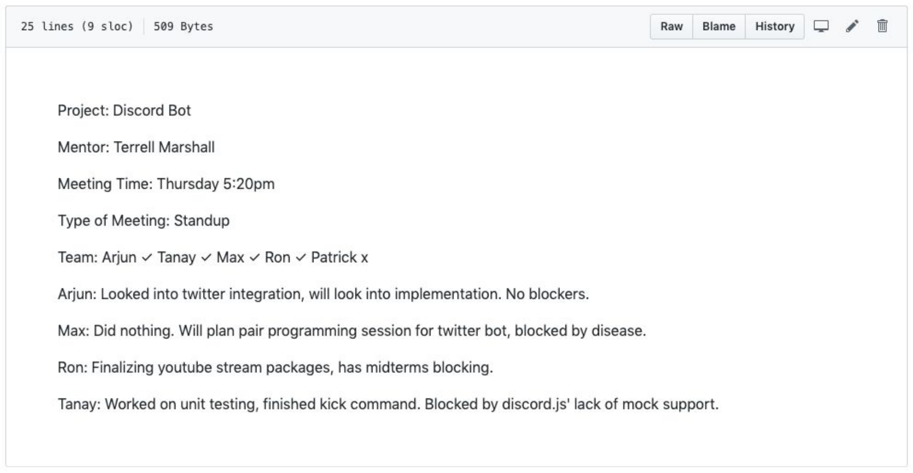

5 lines (4 sloc) | 462 Bytes Raw Blame History

Journey #1: Server owner finds JanetBotpolitano and is satisfied by the listed features. They decide to add it to the server to aid in their moderation journey. In the end, their friends fear their newfound moderation power.

Journey #2: Server owner wants to make it more fun, and finds JanetBotpolitano. They think the minigames it has look fun, and add it to their server. Now that the server owner's server is fun, their friends are never offline.

This helped us look at the project from a different perspective and identify what tools and commands would be most helpful to users of our product.

Then, we had isolated directories for each of our sprints to keep each one's notes and feedback separate from one another. Each sprint directory contained each stand-up's meeting minutes. As an example, here are our stand up notes from our 4th sprint (each sprint was a week long):


25 lines (9 sloc) | 509 Bytes Raw Blame History

Project: Discord Bot

Mentor: Terrell Marshall

Meeting Time: Thursday 5:20pm

Type of Meeting: Standup

Team: Arjun ✓ Tanay ✓ Max ✓ Ron ✓ Patrick x

Arjun: Looked into twitter integration, will look into implementation. No blockers.

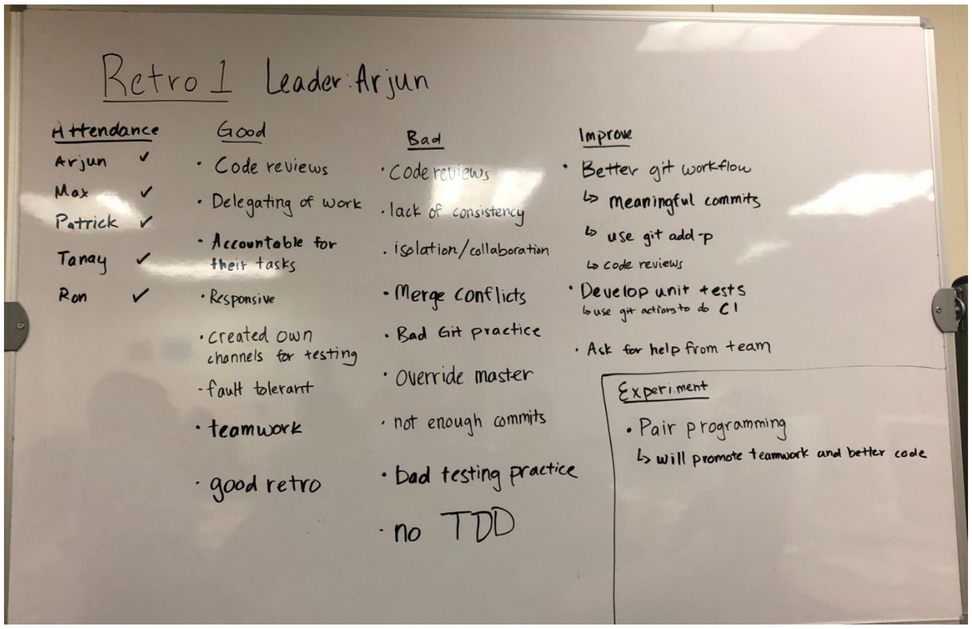
Max: Did nothing. Will plan pair programming session for twitter bot, blocked by disease.

Ron: Finalizing youtube stream packages, has midterms blocking.

Tanay: Worked on unit testing, finished kick command. Blocked by discord.js' lack of mock support.

This allowed us to better gauge how much time was spent on the development of features. We were able to use these notes during our retrospectives. This allowed us to better reflect upon practices that benefited us and practices that were detrimental to our success in our prior sprint.

Finally, we had retrospective repositories, distinct from our source code and sprint notes repository (the main repo), in which we documented analyses and future plans from retros. These retros had a scribe documenting on a whiteboard or iPad; here is an example of one such retro's notes.



Retro 1 Leader: Arjun

Attendance
- Arjun ✓
- Max ✓
- Patrick ✓
- Tanay ✓
- Ron ✓

Good
- Code reviews
- Delegating of work
- Accountable for their tasks
- Responsive
- created own channels for testing
- fault tolerant
- teamwork
- good retro

Bad
- Code reviews
- lack of consistency
- isolation/collaboration
- Merge conflicts
- Bad Git practice
- Override master
- not enough commits
- bad testing practice
- no TDD

Improve
- Better git workflow
  - ↳ meaningful commits
  - ↳ use git add -p
  - ↳ code reviews
- Develop unit tests
  - ↳ use git actions to do CI
- Ask for help from team

Experiment
- Pair programming
  - ↳ will promote teamwork and better code

For full documentation of all our sprints and retros, please refer to the respective repositories.

Implementation difficulties:

When starting, as this was most of our first time's working in detail with Javascript and Node, we struggled a little bit. However, once we got over the initial hump and had a general understanding of the new technologies we were working with, we were fine. Another challenge was using external APIs and being able to read documentation quickly and effectively to find what you're looking for. For example, while working with the Twitter API, we quickly realized that the documentation spanned hundreds of pages, and identifying which subsections had what we were looking for was a bit of a challenge. However, with some practice and familiarity, this became less of an issue. Finally, especially earlier on, we faced some difficulties with merge conflicts. While some merge conflicts are difficult or impossible to avoid, we did a better job at working on distinct and independent tasks such that the frequency of merge conflicts was much less.

Since we began development on the website following the mvp deadline, we were unable to implement all of our desired features. One of the issues we ran into was understanding how to make use of react routers and Oauth to create a settings dashboard for each user. This was something we lacked prior experience with and as a result delayed the development of the settings dashboard. We were able to solve this issue but unfortunately missed the project deadline.

External Resources:

We made use of many third party libraries and frameworks to create our commands. These include:

- Node.js
- Discord.js
- React
- Tailwind
- Jest
- jService.io API
- Weatherstack API
- Youtube API
- Twitter API
- Twit API Client

Testing

Once we had a viable MVP, we were able to conduct some preliminary user testing. Since Discord bots are limited to interaction through Discord's client and web app, our UX decisions mostly focused on argument parsing and intuitive command calls. Since many of the commands were developed in isolation, there were some inconsistencies between commands. For example, the filter command (?filter) took arguments like so: ?filter argument1 argument2… argumentN, while the poll command (?poll) only accepted arguments within quotations: ?poll "option 1" "option 2"... "option N". Cleaning up syntax discrepancies like that took up a surprising amount of resources, since it meant we had to implement regex for each command and its subcommands.

User testing also surfaced comments about the transparency of certain commands. Since the bot was initially designed as an administration tool, some of the original commands were designed only from the user standpoint of someone with admin privileges. The ban and kick commands didn't originally display the justification or reasoning to normal users, and only left a log entry in the admin-only audit panel. We've since rectified this and made admin commands leave a publicly visible record when they're used.

Unit testing posed a unique challenge, as our reliance on Discord's API for our bot's core functionality set us back significantly since no unit testing or mocking framework existed for Discord. We were able to set up unit testing for our website deployment as an alternative.