PUI Assignment 8 | VideoParty | Animesh Singh

Link to Demo: https://youtu.be/TZr0rlPH_X8

Part 1: Description

VideoParty is a web app that enables friends to watch a movie in sync by allowing them to upload a video file, and perform operations like play & pause on the content which automatically syncs for all viewers. The intended use case for this application would be using it on a video call with friends to be able to watch a movie together. For example, if one person pauses the video, it will be paused for all other friends as well - the video syncs in real-time for all viewers.

Built upon users' mental model of the NetflixParty (now called TeleParty) web app, which allows multiple users to sync Netflix videos in real-time. The differentiating factor for VideoParty is that in this app, users can upload ANY video file they might want to watch in sync with friends.

Target Audience: Children, teenagers, young adults - especially after COVID, since a lot of group gatherings have shifted to online modality. An assumption is that younger audiences would be more open/ tech-savvy enough to adopt an online tool like TeleParty for fun.

Part 2: User Interactions with the web-app

1. Get to know how to use VideoParty
   Interaction Type: Visual information display
   a. The user lands on the home page
   b. They see the gif with familiar faces from Parks&Rec (tv show), connotes feelings of warmth, friendship, happiness
   c. They see the infographic explaining the 3-step process in an easy way

2. Upload video file to generate a shareable link
   Interaction Type: HTML form input with button for browsing and uploading a video file onto the Apache server, Apache server recording the upload, putting it in appropriate folder for Golang server to use
   a. The user sees the white box on the right, denotes a call-to-action
   b. They see the "browse" and "upload video" buttons, can connect it to upload mechanism design patterns they've seen on other websites
   c. They click "browse" button and select the video file from their local computer
   d. They click "upload video" button to upload to the server
   e. They get a link which they share with their friends

3. Watch the video in sync with friends
   Interaction Type: AngularJS app with a video player interface, video synced for all players by Golang server
   a. The user sees a familiar video player interface, can connect it to similar video player design patterns on other websites
   b. They start streaming the video by pressing the “play” button
   c. They can pause the video by pressing the “pause” button
   d. Their video stream is synced with their friends

Part 3: External Tools Used

1. AngularJS:

   Why: I wanted to learn a new MVC, and we had learnt React in class, so for the sake of freshness, I decided to learn more about AngularJS.
   How: I used AngularJS to host the Client Side interface for the syncable video player in my project.
   What: To run the project, I learnt how to use **npm** (node package manager) for javascript on my Ubuntu (Linux) system.

   I made changes to the **src/app/app.component.ts** file

```
onPlayerReady(api: VgApiService) {
  this.api = api;
  console.log(this.api.getDefaultMedia().subscriptions);
  interval(100).subscribe(x => {
    this.findStatus();
  });
}
```

   The client side asks for status from server every **100ms**. Increasing this frequency results in many requests to the server, leading it to become slow. Decreasing this frequency causes the video to sync slowly, in an unacceptable manner. Hence, after careful experimentation I decided on this value as a middleground considering the tradeoffs.

```
takeClick(){
  // console.log(this.api.state);
  if (this.api.state === "paused") {
    this.statusService.setStatus("play",this.api.currentTime).subscribe();
  } else {
    this.statusService.setStatus("pause",this.api.currentTime).subscribe();
  }
}
```

Client side toggles status when clicked on play/ pause button.

```
findStatus() {
  // console.log(this.api.state);
  this.statusService.getStatus().subscribe((data: State) => {
    if (this.api.state == "playing" && data.status == "pause") {
      this.api.pause();
      // this.api.seekTime(data.time, false);
    } else if (this.api.state == "paused" && data.status == "play") {
      // this.api.seekTime(data.time, false);
      this.api.play();
    } else if (this.api.state == "paused" && data.status == "paused" &&
      this.api.currentTime != data.time) {
      // this.api.seekTime(data.time, false);
    }
  });
}
```

Client side adjusts its status according to the value of the state it gets from the server.

I also made a file **src/app/status.service.ts** for handling setting and getting status values from server side

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Statement } from '@angular/compiler';

export interface State {
  status: string;
  time: number;
}

@Injectable()
export class StatusService {
  constructor(private http: HttpClient) { }

  statusUrl = 'api/status';

  getStatus() {
    return this.http.get<State>(this.statusUrl);
  }

  setStatus(status: string, time: number) {
    return this.http.post<State>(this.statusUrl, {status: status, time: time});
  }
}
```

2. Go Server:

   Why: I wanted to learn Go, and I learnt that it can be used to make a executable that can serve as a server for this project's purpose. Hence, I decided to use it.
   How: I used a Go server for handling server side functionalities for the syncable video player in my project.
   What: I created the **/go-party-server/server.go** file for handling the server side of syncing videos.

```
func GetStatus(ec echo.Context) error {
    mu.Lock()
    defer mu.Unlock()
    return ec.JSON(http.StatusOK, state)
}

func SetStatus(ec echo.Context) error {
    reqState := new(State)
    if err := ec.Bind(reqState); err != nil {
        return err
    }
    mu.Lock()
    state = *reqState
    defer mu.Unlock()
    return ec.JSON(http.StatusNoContent, nil)
}
```

   These functions help handle the **setting and getting status** function calls from the multiple client side(s).

3. PHP, HTTP Server through Python3/ Apache/ XAMPP:

   Why: There was a need to handle the uploading aspect of the user-flow for making VideoParty work. I enjoyed learning PHP and how to host local servers.
   How: I created a local Apache server (using XAMPP on Windows and Python3 on

Ubuntu) to handle the uploaded video from the user's side on the home page.
What: The server helps to retrieve the video file from the user's personal file directory and put it into the server's file directory so that it can be hosted to the multiple clients.

```php
if ($uploadOk == 0) {
  echo "Sorry, your file was not uploaded.";
// if everything is ok, try to upload file
} else {
  if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {
    echo "The file ". htmlspecialchars( basename( $_FILES["fileToUpload"]["name"])). " has been uploaded. Here's your
    link ==> ". "<a href='". $link_address."' target='_blank'>Link</a>";
  } else {
    echo "Sorry, there was an error uploading your file.";
  }
}
?>
```

Handling video file upload through PHP.

4. Bootstrap:

   Why: I wanted to use and learn more about Bootstrap functionalities.
   How: I used the in-build **12-column layout** from Bootstrap for the homepage of VideoParty.
   What: The 12-column layout is standard industry practice for designing responsive websites. Using this feature helped me make my web-app **responsive**.

Part 4: Iteration on A7 Design

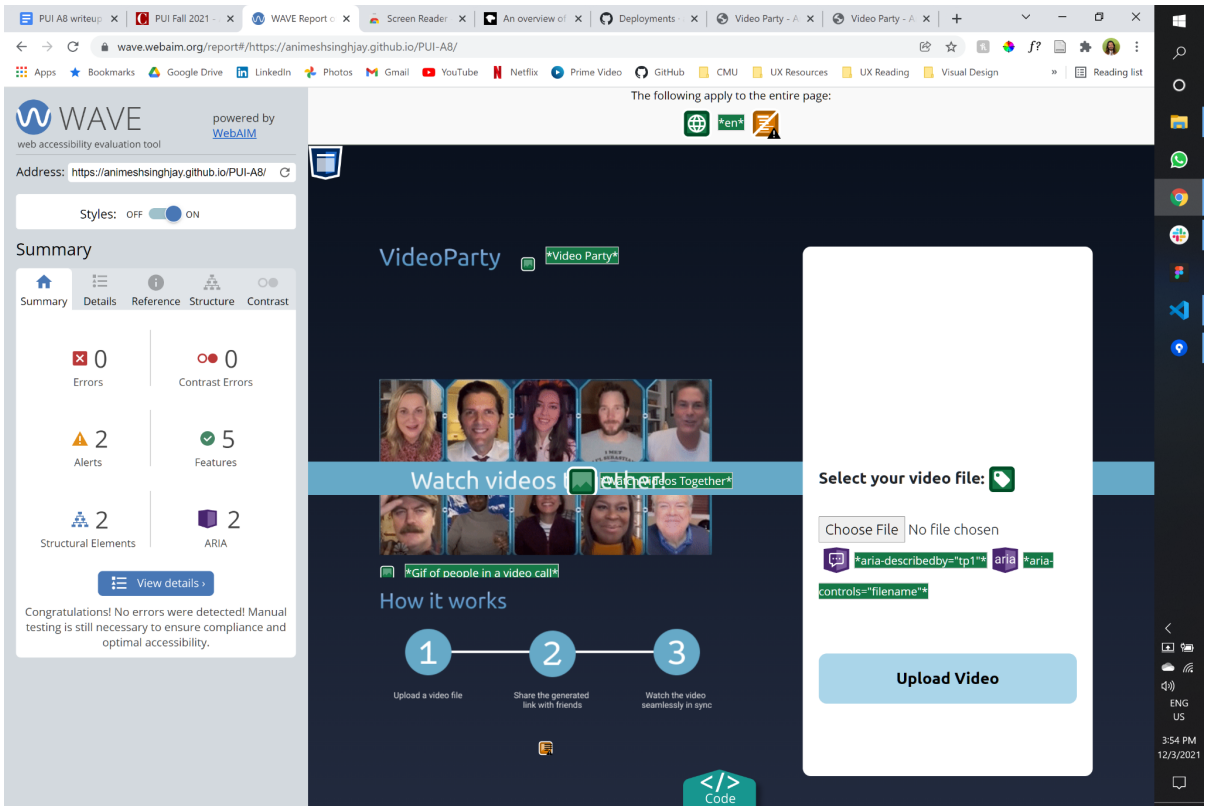
I dropped the idea of using Amazon AWS, and decided to implement my project using locally hosted servers. I used a Go server for server side request handling and AngularJS for client side requests and visual interface. I also used a local Apache server for handling the upload video file functionality.

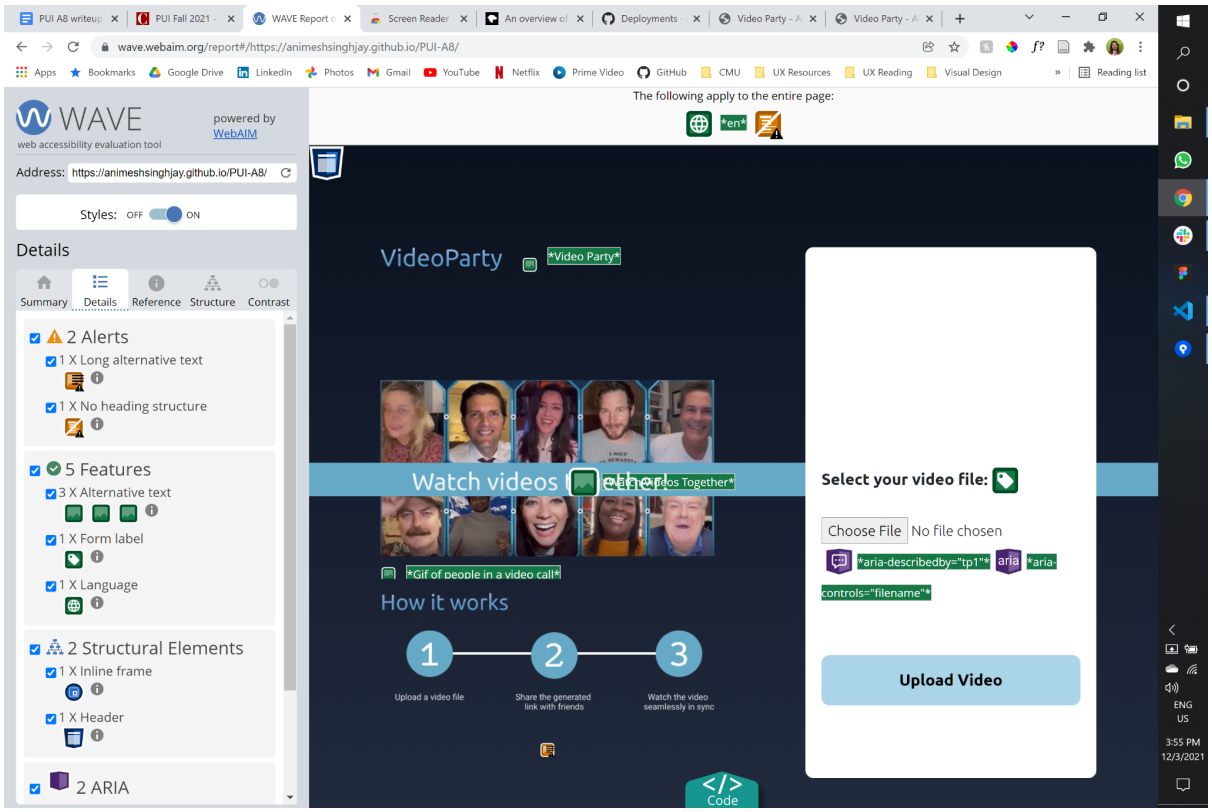
Part 5: Challenges

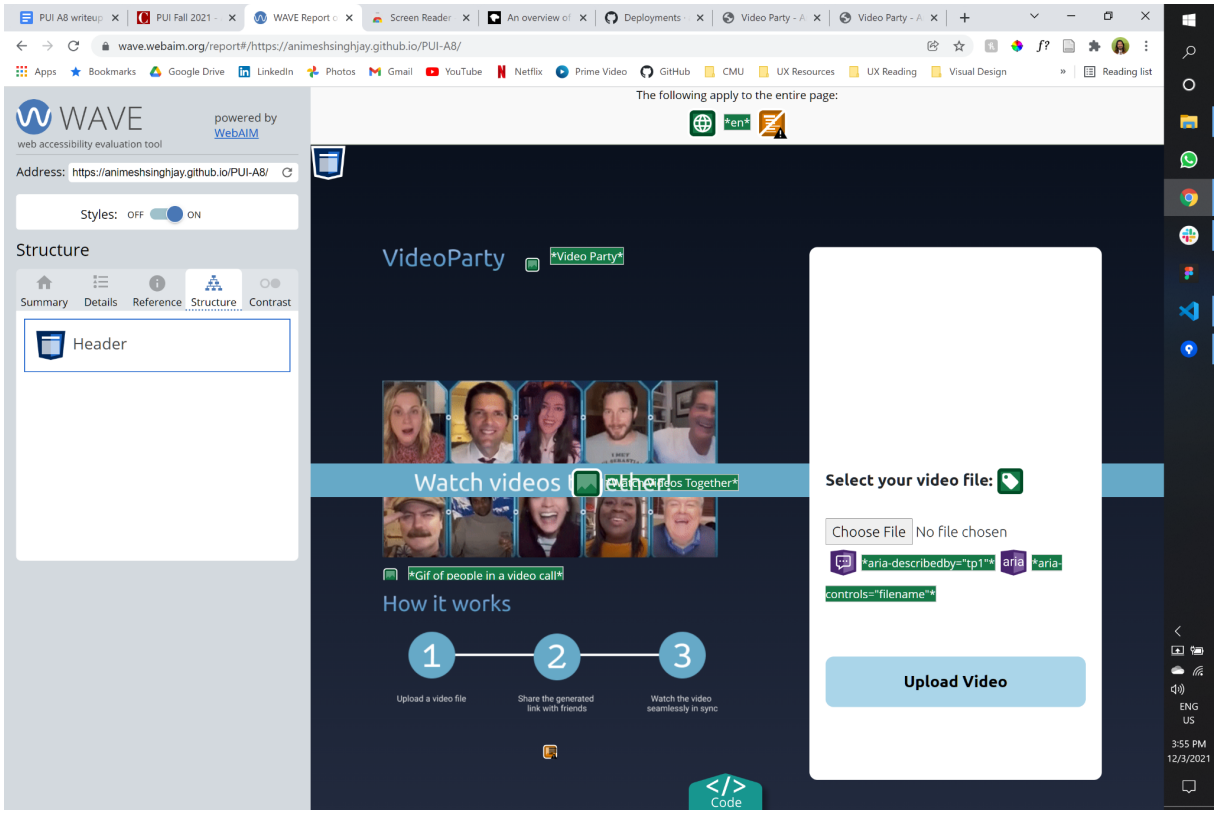
I knew that this was going to be a challenging project, and I admittedly faced a number of roadblocks - figuring out how to get all these new dependencies that I was using to work, figuring out how to run servers, switching to Linux when I was not able to figure out how to get stuff to work on Windows. I was able to overcome these (after hours of googling, trying things out, and debugging), and was finally able to get a working prototype ready.

Responsive Design + WAVE Tool for Accessible Design

The web app is responsive as it is based on Bootstrap's 12-column layout. It resizes when the viewport size is changed. The website also follows accessibility standards, has passed the test from the WAVE web accessibility evaluation tool.







Extra Credit

I added compatibility for screen readers by following the ARIA guidelines.
Link to Screen Reader Demo: https://youtu.be/qnxfM2jdKaQ

Demo Video Disclaimer

The demo video shows the initial part of the user flow (uploading video onto the web app) on a Windows interface. Then the latter part of the user flow (video player, syncing of video) is on an Ubuntu interface. The reason for this is that I first figured out the upload part when I was using Windows (by setting up a local Apache server using XAMPP), and later (for the sake of keeping my sanity) switched to Ubuntu for the video sync part because it involved setting up multiple servers (Go, AngularJS, Python). I couldn't figure out how to perform the video upload on Ubuntu because of the lack of error messages from PHP, and the local Apache server (and it having a billion dependencies). Anyway, I felt that it was not a super important aspect, so I decided not to spend more time figuring out the Apache server on Ubuntu, as I had already done it on Windows. Hence the only assumption is that the video file is taken from the user's directory and put into the /uploads directory. The python3 http server is set up in this directory which hosts the video on localstorage.