#Low5Selfie Challenge

There are not enough selfies of low fives in the world. Aspiring IG influencers are looking for unique tools to create quality content that will differentiate their feed, grow their follower count, and get them the blue checkmark.

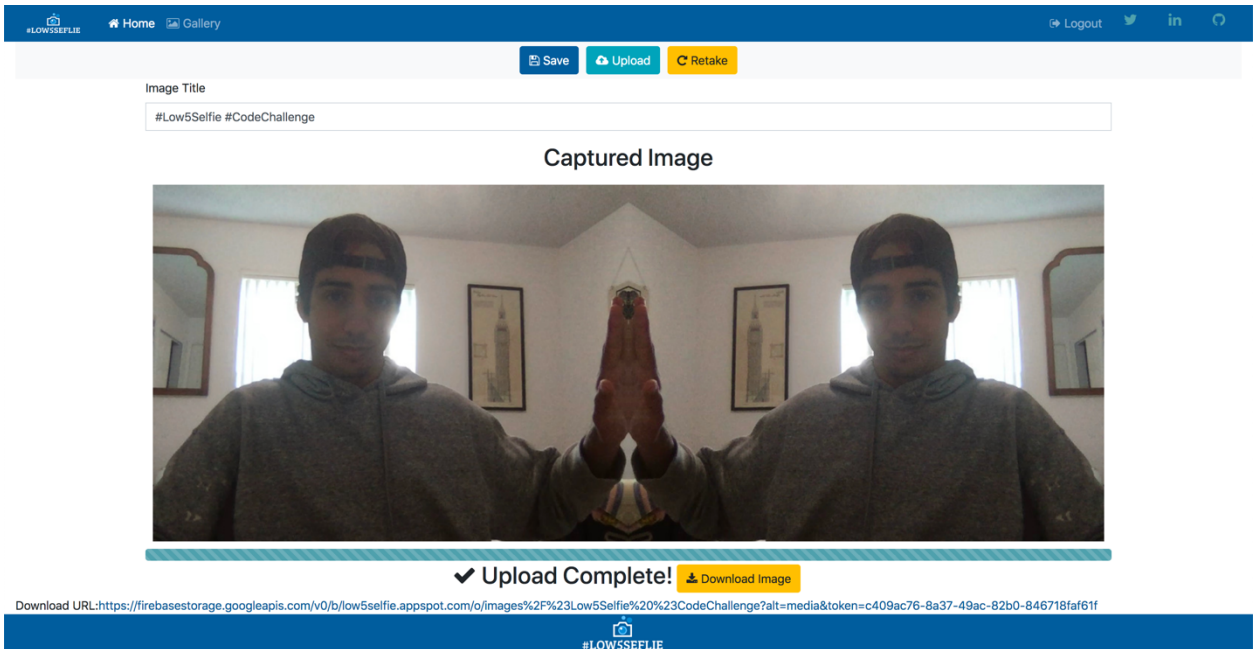


Table of Contents

Code Challenge

Problem
There are not enough selfies of low fives in the world. Aspiring IG influencers are looking for unique tools to create quality content that will differentiate their feed, grow their follower count, and get them the blue checkmark.

Appetite
The constraints for building a solution to this problem is 3 days time with a team of 1 designer/developer.

Solution
Craft a unique experience that makes it super easy to take a selfie while giving yourself a low five. #Low5Selfie

Aspiring IG influencers should be able to…

- Visit a single page web app
  - developed in Angular with TypeScript
  - open-source components are cool
- Grant permission to their camera
  - within the web browser
- View their camera's video stream
  - within a 2-column, side-by-side, layout
  - utilizing a majority of the page
  - styled/transformed in a way that aligns for a #Low5Selfie
- Click a button to capture the #Low5Selfie as a screenshot
  - use Pico, an open-source screenshot library
  - other open-source libraries are cool too
- Upload the #Low5Selfie image to a backend endpoint
  - frontend implemented as an Angular service
  - backend developed in Node with Express
  - open-source Node packages are cool
  - the request body should include the selfie
  - storing the #Low5Selfie image in the local file system is cool
  - the response body should include the URL of the #Low5Selfie image
- View the #Low5Selfie image's URL
- Take another #Low5Selfie

Bonus...

- Switch between cameras, updating the camera stream
  - implemented as an observable
- Preview the selfie before uploading to the cloud
- Display selfie upload progress
- Secure the cloud upload endpoint with a token
- Validate the token with Express middleware

Super Bonus!!!

- Present the app with a beautiful #Low5Selfie brand
- Deploy both apps to App Engine
    - no fancy domains necessary
    - the Google Cloud Platform trial period is free

Background

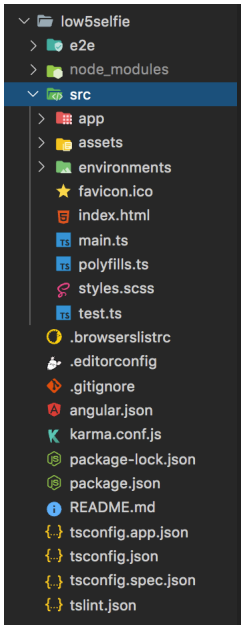
The main technology stack used to complete this challenge was MEAN stack. Angular, Express, Node, and MongoDB. Later we will deploy to Firebase to host the app and have an online cloud storage. The main goal is to be able to take a picture using the users webcam and mirroring the results so that the user can take a Low5Selfie! Along with the MEAN stack we'll be using simple SCSS styling and Bootstrap to layout the site.

Process & Implementation

I. Initial Setup

First steps are to install the latest version of the tools. Assuming Node is already installed we can make a new directory for our project. First, we will generate our front-end by using the Angular CLI:

```
ng new low5selfie
```



This will generate a new Angular Project and fill the directory with the pre-constructed files needed. When promoted say **Yes** for angular routing and choose **SCSS** styling for the style. The structure should look like the figure on the right. We can check to see our app runs properly by running:

```
ng serve
```

This will start our app on the **localhost:4200** port by default. Next we can go ahead and start installing some dependences we need. We will need a few to complete our task I will go into more detail once we start to use them. First we need to run the following command:

```
npm install --save @ng-bootstrap/ng-bootstrap bootstrap express core-
js file-saver font-awesome ngx-webcam ts-image-processor firebase
angularfire2
```

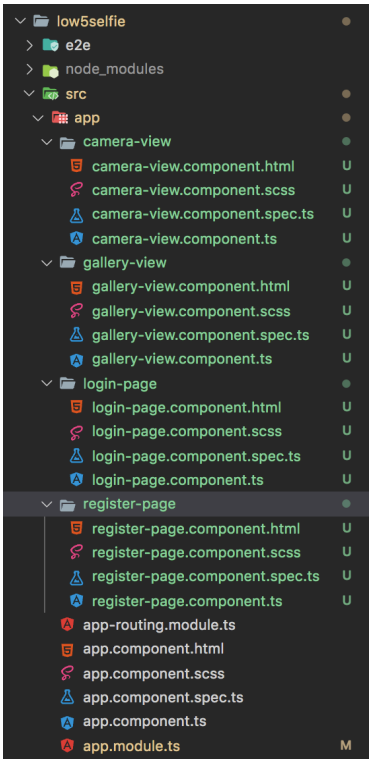
This will install all the dependences we will use throughout the project. Next we can start to create the components we will use in the project using the Angular CLI boilerplate component generator. We need four components:

- **CameraComponent** – This is the main camera view component.

- **LoginComponent** – This is the Login Page component to allow a user to login to the site to see the Gallery and Upload Images.
- **RegisterComponent** – This is the Register Page Component to allow a user to Sign-up to the site.
- **GalleryComponent** – This component will show all the photos available on the MongoDB and display them.

From the low5selfie directory we can generate these components by running the following commands:

```
ng generate component camera-view
ng generate component login-page
ng generate component register-page
ng generate component gallery-view
```



This will then generate the components we need and automatically include them in our angular modules. The folder structure should look a bit like the figure on the right. Now that we have the boilerplate components, we can start to fill in the blanks.

II. Front-End

First Step now is the build a simple navbar using Bootstrap on the **app.component.html** so that we have a nice fluid layout for the app. For the purpose of this I will simply include all the **SCSS** that will be used by the app to make our lives easier.

Let's start by removing the boilerplate HTML markup that is created by the Angular CLI. The goal here is to create a **navbar** and our route links that will be connected later on to angular-routes. Feel free to replace any **href** links with your personal social links and github. The **../assets/logo.png** should also be replaced with your own personal logo. There is several angular directives used here to set views depending on the other components. Few directives I'd like to point out is the:

- ***ngIf="_authService.loggedIn()" :** This directive is provided by the auth services that will be created later to verify if a User is logged in. By using***ngif** we can display or hide this HTML element.
- **(click)="_authService.logout()" :** This directive will execute the logout method for the user to be logged out when they're done using the app.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
    <a class="navbar-brand" href="#">
        <img src="../assets/logo.png" height="90" width="100" class="d-inline-block
align-top" alt="" />
    </a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-
expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item">
                <a class="nav-link" routerLink="/home" routerLinkActive="active"><i
class="fa fa-home"></i>
          Home
          <span class="sr-only">(current)</span></a
        >
      </li>
      <li class="nav-item">
        <a class="nav-link" routerLink="/gallery" routerLinkActive="active"
          ><i class="fa fa-image"></i> Gallery</a
        >
      </li>
    </ul>
    <ul class="navbar-nav ml-auto">
      <li class="nav-item">
        <a
          class="nav-link"
          *ngIf="!_authService.loggedIn()"
          routerLink="/register"
          routerLinkActive="active"
          ><i class="fa fa-user-plus"></i> Register</a
        >
      </li>
      <li class="nav-item">
        <a
```

1 - app.component.html

```
          class="nav-link"
          *ngIf="!_authService.loggedIn()"
          routerLink="/login"
          routerLinkActive="active"
          ><i class="fa fa-sign-in"></i> Login</a
        >
      </li>
      <a
        class="nav-link"
        style="cursor: pointer"
        *ngIf="_authService.loggedIn()"
        (click)="_authService.logoutUser()"
        ><i class="fa fa-sign-out"></i> Logout</a
      >
      <li class="nav-item">
        <a
          class="btn-floating btn-lg btn-outline-secondary"
          type="button"
          role="button"
          ><i class="fa fa-twitter"></i
        ></a>
            </li>
            <li class="nav-item">
                <a class="btn-floating btn-lg btn-outline-secondary" type="button"
role="button" href="https://www.linkedin.com/in/jeancarlosp/"><i class="fa fa-
linkedin"></i
        ></a>
            </li>
            <li class="nav-item">
                <a class="btn-floating btn-lg btn-outline-secondary" type="button"
role="button" href="https://github.com/Jpere547/Low5Selfie-CodeChallenge"><i class="fa
fa-github"></i
        ></a>
            </li>
        </ul>
    </div>
</nav>
```

```
<div class="container">
    <router-outlet> </router-outlet>
</div>
<nav class="navbar navbar-expand-lg navbar-dark bg-primary align-bottom justify-
content-center">
    <a class="navbar-brand" href="#">
        <img src="../assets/logo.png" height="140" width="140" class="d-inline-block
align-top" alt="" />
    </a>
</nav>
```

In order for our HTML to render properly we need to create the **auth providers and service**. Angular CLI once again can generate this for us using the command:

```
ng generate guard auth
```

We should see a few options to choose from when running this command. For this project we will want to select the **CanActivate** option from the CLI. This will generate two new files **auth.guard.ts** and **auth.guard.spec.ts**. In order to keep a clean file structure, we can go head and make new directory in **src/** and name it **auth/**. And move our newely generated guard files in there. *Don't forget to edit the path in **app.module.ts**.* Next we're gonna want to create an Interceptor in order to let our application make an HTTP request using the **HttpClient** service which will call the **intercept()** method. Within the **src/auth/** directory we can create three new files:

- **token-interceptor.service.ts :** This will allow us to use the **HttpInterceptor** to handle the HTTP requests from the application on the global level. Along with sending the generated Token to the request header.
- **event.service.ts :** This file will be the ImageService Injectable to handle the requests to view the gallery when a user is logged in.
- **auth.service.ts :** This file will handle the Injectable routers to handle requests for logging in a user, registering a user, and posting new images.

Below is a preview of all the auth files and their contents:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private _authService: AuthService,
    private _router: Router) { }

  canActivate(): boolean {
    if (this._authService.loggedIn()) {
      console.log('true')
      return true
    } else {
      console.log('false')
      this._router.navigate(['/login'])
      return false
    }
  }
}
```

2. auth.guard.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Router } from '@angular/router';

@Injectable()
export class AuthService {
  private _registerUrl = 'http://localhost:3005/server/register';
  private _loginUrl = 'http://localhost:3005/server/login';
  private _imagePostUrl = 'http://localhost:3005/server/gallery';

  constructor(private http: HttpClient, private _router: Router) {}

  registerUser(user) {
    return this.http.post<any>(this._registerUrl, user);
  }

  loginUser(user) {
    return this.http.post<any>(this._loginUrl, user);
  }

  logoutUser() {
    localStorage.removeItem('token');
    this._router.navigate(['/home']);
  }

  postImage(image) {
    return this.http.post<any>(this._imagePostUrl, image);
  }

  getToken() {
    return localStorage.getItem('token');
  }

  loggedIn() {
    return !!localStorage.getItem('token');
  }
}
```

3. auth.service.ts

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

@Injectable()
export class ImageService {
  private _galleryUrl = 'http://localhost:3005/server/gallery';

  constructor(private http: HttpClient) {}

  getImages() {
    return this.http.get<any>(this._galleryUrl);
  }
}
```

4. event.service.ts

```
import { Injectable, Injector } from '@angular/core';
import { HttpInterceptor } from '@angular/common/http'
import { AuthService } from './auth.service';
@Injectable()
export class TokenInterceptorService implements HttpInterceptor {

  constructor(private injector: Injector){}
  intercept(req, next) {
    let authService = this.injector.get(AuthService)
    let tokenizedReq = req.clone(
      {
        headers: req.headers.set('Authorization', 'bearer ' +
authService.getToken())
      }
    )
    return next.handle(tokenizedReq)
  }

}
```

5. token-interceptor.service.ts

Before viewing our changes we need to include a few modules into our **app.module.ts:** By including these changes and importing our dependencies we installed we can start to build the main components. We also need to add the routing to the **app-routing.modules.ts** file to handle which components will be render given a route.

```
import { BrowserModule } from '@angular/platform-browser';
import { AuthGuard } from './auth/auth.guard';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { environment } from '../environments/environment';
import { AuthService } from './auth/auth.service';
import { ImageService } from './auth/event.service';
import { TokenInterceptorService } from './auth/token-interceptor.service';
import { WebcamModule } from 'ngx-webcam';

import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
import { AngularFireModule } from '@angular/fire';
import { AngularFireStorageModule } from '@angular/fire/storage';
import { CameraViewComponent } from './pages/camera-view/camera-
view.component';
import { LoginPageComponent } from './pages/login-page/login-page.component';
import { RegisterPageComponent } from './pages/register-page/register-
page.component';
import { GalleryPageComponent } from './pages/gallery-page/gallery-
page.component';

@NgModule({
  declarations: [
    AppComponent,
    CameraViewComponent,
    LoginPageComponent,
    RegisterPageComponent,
    GalleryPageComponent,
  ],
  imports: [
    BrowserModule,
```

```
    AppRoutingModule,
    NgbModule,
    WebcamModule,
    FormsModule,
    HttpClientModule,
    AngularFireStorageModule,
    AngularFireModule.initializeApp(environment.firebaseConfig, 'cloud'),
  ],
  providers: [
    AuthService,
    AuthGuard,
    ImageService,
    {
      provide: HTTP_INTERCEPTORS,
      useClass: TokenInterceptorService,
      multi: true,
    },
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

6. app.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AuthGuard } from './auth/auth.guard';
import { CameraViewComponent } from './pages/camera-view/camera-
view.component';
import { GalleryPageComponent } from './pages/gallery-page/gallery-
page.component';
import { LoginPageComponent } from './pages/login-page/login-page.component';
import { RegisterPageComponent } from './pages/register-page/register-
page.component';

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: CameraViewComponent },
  { path: 'login', component: LoginPageComponent },
  { path: 'register', component: RegisterPageComponent },
  {
    path: 'gallery',
    canActivate: [AuthGuard],
    component: GalleryPageComponent,
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

7. app-routing-module.ts

For the time being we can comment out the **Firebase Modules** as we won't be using them just yet. Last thing we need is to add our **_authService** to the AppComponent constructor.

```
constructor(public _authService: AuthService) {}
```

Next we can add the initial styling and import **Bootstrap** and **Font-Awesome** into the styles.scss:

```
/* You can add global styles to this file, and also import other style files
*/

$primary: #1C5D99;
$secondary: #639FAB;
$success: #2828dc;
$danger: #222222;
@import "../node_modules/bootstrap/scss/bootstrap.scss";
@import "~font-awesome/scss/font-awesome.scss";
body,
html {
    height: 100%;
}

.btn,
.btn-floating {
    margin: 5px;
}

.navbar-expand-lg {
    max-height: 50px;
}

.container {
    max-width: 1980px;
}

img {
    padding: 10px !important;
}
```

Given these styles we can move on the main course of the front-end app the **CameraViewComponent.**

This component will hold the heart of our app. In order to get a camera, view we will be using the **ngx-camera** package. This package allows us to ask permission to use the user's camera from the web app and use their observable functions to capture event triggers when the user wants to capture a picture. As well as mirror the view so that we can give ourselves a low5selfie. I will break down each function used in the CameraViewComponent:

- **toggleWebcams():** This function is a simple Boolean that will toggle the camera on or off depending on the user button event click.
- **toggleNextWebcam():** This function will toggle to the next webcam if the user has more than one webcam available.
- **toggleRetake():** This function will allow a user to retake a photo after a capture event was triggered. It resets the Boolean values needed to refresh the component back to the default state.
- **triggerSnapshots():** This function will be the trigger for the Obserable when its fired the image will be captured and emitted.
- **imageEventHandler(webcamSnapshotImg: WebcamImage):** This function will handle the snapshot from the triggerSnapshots(). Here the image will be captured into a webcameSnapshotImg variable.
- **imageToBlob(imageBase64: string):** Since the **ngx-webcam** package creates a Base64 image type this function will return an Blob Image type given the inputted image.
- **combImage(image1: string, image2: string):** This function will combined the two images. The trick here is that we use **ts-image-processor** package to flip the second image. Since ngx-webcam takes two pictures even if the second view is mirrored, they're still captured in the same rotation. This work around can be used to flip the second image to give it the mirrored effect.
- **saveLocally():** This function will save an image locally using the **file-saver** package. It will save the image as "image.png".
- **downloadImage():** This function is very similar to the above function but the information is pulled from Firebase rather than the **webcamSnapshotImg** .
- **uploadToDB():** This function will use the auth.service to send a post request to the backend to store the **imageData** including the Image Title and Image URL.
- **uploadFirebase():** Similar to the above function this will issue a Firebase Promise to upload the image to the Firebase Storage Cloud.
- **get triggerSnapshotObservable():** This is the Observable RxJs used to trigger the image capturing when fired. **triggerSnapshots()**
- **get triggerNextWebcam():** This is the Observable RxJs used to trigger the next available webcam **toggleNextWebcam()**

Our **camera-view.component.ts** will look something like:

```
import { Component, OnInit } from '@angular/core';
import { WebcamImage, WebcamInitError, WebcamUtil } from 'ngx-webcam';
import { Subject, Observable } from 'rxjs';
import { finalize } from 'rxjs/operators';
import {
  AngularFireStorage,
  AngularFireUploadTask,
} from '@angular/fire/storage';
import { AuthService } from '../../auth/auth.service';
import { Router } from '@angular/router';
import { saveAs } from 'file-saver';
import mergeImages from 'merge-images';
import { imageProcessor, mirror } from 'ts-image-processor';

@Component({
  selector: 'app-camera-view',
  templateUrl: './camera-view.component.html',
  styleUrls: ['./camera-view.component.scss'],
})
export class CameraViewComponent implements OnInit {
  constructor(
    private storage: AngularFireStorage,
    private _auth: AuthService,
    private _router: Router
  ) {}

  private nextCamera: Subject<boolean> = new Subject<boolean>(); // Subject
Object for Swapping Webcams
  private snapshotTrigger: Subject<void> = new Subject<void>();
  toggleWebcam = true;
  toggleSnapshotMode = false;
  webErrors: WebcamInitError[] = []; // Capture ngxWebcam errors
  allowCameraSwitch = true;
  webcamSnapshotImg: WebcamImage = null; //Container class for the captured
image
  multipleWebcamsAvailable = false; // Checks if the users has multiple
webcams
```

```
  downloadURL: Observable<string>;
  task: AngularFireUploadTask;
  percentage: Observable<number>;
  isUploadComplete = false;
  mergedImageBlob: Blob;
  mergedImageURL;
  imageData = {
    title: '',
    dataurl: '',
  };

  // On Initialization lists available videoInput devices if more than one is
avaiable
  public ngOnInit(): void {
    WebcamUtil.getAvailableVideoInputs().then((devices: MediaDeviceInfo[]) =>
{
      this.multipleWebcamsAvailable = devices && devices.length > 1;
    });
  }

  // Toggle Function to turn on or off the webcam
  public toggleWebcams(): void {
    this.toggleWebcam = !this.toggleWebcam;
  }

  // Toggle Function to Rotation between cameras
  public toggleNextWebcam(): void {
    this.nextCamera.next(true);
  }

  // Toggle Retake Picture
  public toggleRetake(): void {
    this.webcamSnapshotImg = null;
    this.toggleSnapshotMode = false;
    this.isUploadComplete = false;
    this.downloadURL = null;
  }
```

```
// Toggle function to take snapshot
public triggerSnapshots(): void {
  this.snapshotTrigger.next();
}

// Event Handler Function for when an Image Snapshot is taken
public imageEventHandler(webcamSnapshotImg: WebcamImage): void {
  this.webcamSnapshotImg = webcamSnapshotImg;
  this.toggleSnapshotMode = true;
  this.combImage(
    this.webcamSnapshotImg.imageAsDataUrl,
    this.webcamSnapshotImg.imageAsDataUrl
  );
}

// Convert Base64 WebcamImage to File for Local Save
public imageToBlob(imageBase64: string) {
  imageBase64 = imageBase64.replace(/^[^,]+,/, '');
  const byteString = window.atob(imageBase64);
  const arrayBuffer = new ArrayBuffer(byteString.length);
  const int8Array = new Uint8Array(arrayBuffer);
  for (let i = 0; i < byteString.length; i++) {
    int8Array[i] = byteString.charCodeAt(i);
  }
  const imageBlob = new Blob([int8Array], { type: 'image/png' });
  this.mergedImageBlob = imageBlob;
  let reader = new FileReader();
  reader.readAsDataURL(imageBlob);
  reader.onloadend = () => {
    this.mergedImageURL = reader.result;
  };
}

// Combine Image and Mirrored Together
public combImage(image1: string, image2: string) {
  imageProcessor
    .src(image2)
    .pipe(mirror())
    .then((mirroredImg) => {
```

```
      mergeImages(
        [
          { src: image1, x: 0, y: 0 },
          { src: mirroredImg, x: 640, y: 0 },
        ],
        { width: 1280, height: 480 }
      )
        .then((b64: string) => {
          this.imageToBlob(b64);
        })
        .catch((err) => console.log(err));
    });
}

// Save Image File Locally
public saveLocally(): void {
  const image = this.mergedImageBlob;
  console.log(image);
  saveAs(image, 'image.png');
}

// Upload to MongoDB
public uploadToDB(url: string): void {
  this.imageData.dataurl = url;
  this._auth.postImage(this.imageData).subscribe(
    (res) => {
      localStorage.setItem('token', res.token);
    },
    (err) => console.log(err)
  );
}

// Download from Firebase
public downloadImage(): void {
  saveAs(this.downloadURL, 'image.png');
}

// Upload to Firebase
public uploadFirebase(): void {
  const file = this.mergedImageBlob;
```

```
    const path = `images/${this.imageData.title}`;
    const storageRef = this.storage.ref(path);
    this.task = storageRef.put(file);
    this.percentage = this.task.percentageChanges();
    this.task
      .snapshotChanges()
      .pipe(
        finalize(() => {
          this.isUploadComplete = true;
          this.storage
            .ref(path)
            .getDownloadURL()
            .subscribe((url) => {
              this.downloadURL = url;
              this.uploadToDB(url);
            });
        })
      )
      .subscribe();
  }

  // Creates a new Observable with snapshotTrigger as the source.
  public get triggerSnapshotObservable(): Observable<void> {
    return this.snapshotTrigger.asObservable();
  }

  // Creates a new Observable with nextCamera as the source.
  public get triggerNextWebcame(): Observable<boolean | string> {
    return this.nextCamera.asObservable();
  }
}
```

In order to view our new functions we need to add the HTML to the **camera-view.compoent.html** this should look like:

```
<nav
  class="navbar navbar-expand-lg navbar-dark bg-light justify-content-md-
center"
  *ngIf="!toggleSnapshotMode"
```

```
>
  <div class="row">
    <button
      class="btn btn-primary"
      (click)="triggerSnapshots()"
      [disabled]="!toggleWebcam"
    >
      <i class="fa fa-camera-retro"></i>
      Capture
    </button>
    <button
      class="btn btn-info"
      [disabled]="!multipleWebcamsAvailable"
      (click)="toggleNextWebcam()"
    >
      <i class="fa fa-recycle"></i>
      Swap Cameras
    </button>
    <button
      class="btn btn-danger"
      (click)="toggleWebcams()"
      *ngIf="!toggleWebcam"
    >
      <i class="fa fa-toggle-off"></i>
      Camera On
    </button>
    <button
      class="btn btn-danger"
      (click)="toggleWebcams()"
      *ngIf="toggleWebcam"
    >
      <i class="fa fa-toggle-on"></i>
      Camera Off
    </button>
  </div>
</nav>
<nav
  class="navbar navbar-expand-lg navbar-dark bg-light justify-content-md-
center"
  *ngIf="toggleSnapshotMode"
```

```
>
  <div class="row justify-content-md-center">
    <button
      class="btn btn-primary md-col-4"
      [disabled]="!imageData.title"
      (click)="saveLocally()"
    >
      <i class="fa fa-save"></i>
      Save
    </button>
    <button
      class="btn btn-info md-col-4"
      [disabled]="!imageData.title"
      (click)="uploadFirebase()"
    >
      <i class="fa fa-cloud-upload"></i>
      Upload
    </button>
    <button class="btn btn-warning md-col-4" (click)="toggleRetake()">
      <i class="fa fa-repeat"></i>
      Retake
    </button>
  </div>
</nav>

<body class="mg-auto">
  <div class="container-fluid" *ngIf="!webcamSnapshotImg">
    <div class="row cameraRow">
      <div class="cameraView">
        <webcam
          [trigger]="triggerSnapshotObservable"
          (imageCapture)="imageEventHandler($event)"
          *ngIf="toggleWebcam"
          [allowCameraSwitch]="allowCameraSwitch"
          mirrorImage="never"
        >
        </webcam>
      </div>
      <div class="cameraViewMirrored">
        <webcam
```

```
        *ngIf="toggleWebcam"
        [allowCameraSwitch]="allowCameraSwitch"
        mirrorImage="always"
      >
      </webcam>
    </div>
  </div>
  <br />
</div>
<div class="container" *ngIf="webcamSnapshotImg">
  <div class="row justify-content-md-center">
    <div class="snapshot">
      <form class="form display-flex justify-content-center">
        <div class="form-group">
          <label>Image Title</label>
          <input
            class="col-6"
            type="text"
            [(ngModel)]="imageData.title"
            name="title"
            class="form-control rounded-0"
            placeholder="Please Enter An Image Title"
            required
          />
        </div>
      </form>
      <h2 class="text-center">Captured Image</h2>
      <img [src]="mergedImageURL" />

      <div class="progress">
        <div
          class="progress-bar progress-bar-striped bg-secondary"
          role="progressbar"
          [style.width]="(percentage | async) + '%'"
          [attr.aria-valuenow]="percentage | async"
          aria-valuemin="0"
          aria-valuemax="100"
        ></div>
      </div>
      <div class="row justify-content-md-center" *ngIf="isUploadComplete">
```

```
          <h2 class="md-col-6 ml">
            <i class="fa fa-check"></i> Upload Complete!
          </h2>
          <button class="btn btn-warning md-col-6 mr"
(click)="downloadImage()">
            <i class="fa fa-download"></i> Download Image
          </button>
        </div>
      </div>
    </div>
    <div class="row">
      Download URL:<a
        class="justify-content-center text-md-center"
        [href]="downloadURL"
      >
        {{ downloadURL }}</a
      >
    </div>
  </div>
</body>
```

Most of this HTML is using directives from the component module to complete different functionalities of the app. Next we're going to fill in the Login and Register Page. The idea behind these components is to create a simple request using the AuthService to allow a user to register and login if that user has a username and password. We can use the JWT token to verify if a user is unique and authenticate the login request.

login-page.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../../auth/auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login-page',
  templateUrl: './login-page.component.html',
  styleUrls: ['./login-page.component.scss'],
})
export class LoginPageComponent implements OnInit {
  loginUserData = {
    email: '',
    password: '',
```

```
  };

  constructor(private _auth: AuthService, private _router: Router) {}

  ngOnInit() {}

  loginUser() {
    this._auth.loginUser(this.loginUserData).subscribe(
      (res) => {
        localStorage.setItem('token', res.token);
        this._router.navigate(['/home']);
      },
      (err) => console.log(err)
    );
  }
}
```

register-page.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../../auth/auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-register-page',
  templateUrl: './register-page.component.html',
  styleUrls: ['./register-page.component.scss'],
})
export class RegisterPageComponent implements OnInit {
  registerUserData = { email: '', password: '' };
  constructor(private _auth: AuthService, private _router: Router) {}

  ngOnInit() {}

  registerUser() {
    this._auth.registerUser(this.registerUserData).subscribe(
      (res) => {
        localStorage.setItem('token', res.token);
        this._router.navigate(['/home']);
      },
      (err) => console.log(err)
```

```
    );
  }
}
```

These components use the Angular Router to send the post and get request to the Back-End.

The last front end component we need is the gallery view component which will pull from the backend MongoDB to display photos stored in a gallery view to see different users low5selfie. This component will use the **event.service.ts** we created to handle the get Request to view the images and verify they're logged in.

gallery-page.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { ImageService } from '../../auth/event.service';
import { saveAs } from 'file-saver';

@Component({
  selector: 'app-gallery-page',
  templateUrl: './gallery-page.component.html',
  styleUrls: ['./gallery-page.component.scss'],
})
export class GalleryPageComponent implements OnInit {
  images = [];
  constructor(private _imageService: ImageService) {}

  ngOnInit(): void {
    this._imageService.getImages().subscribe(
      (res) => (this.images = res),
      (err) => console.log(err)
    );
  }

  public downloadImage(url): void {
    saveAs(url, 'image.png');
  }
}
```

Now that our front end is complete, we can start to work on the back. Most of the HTML used in the last three components is straight forward. More information on the HTML can be view in the github.

III. Back-End

The Back-End is a lot less code and straight forward. Let's create a new folder outside the **low5selfie** folder and create a new folder we can call it **server/** here we can run a new npm install command to get the needed dependencies:

```
npm install --save express express-validator jsonwebtoken mongoose
cors body-parser
```

These packages can get us up and running quickly and start up our backend server. First we need to create a **server.js** to import the express package to setup our middleware routes.

Server.js

```
const express = require("express");
const bodyParser = require("body-parser");
const cors = require("cors");
const path = require("path");

const api = require("./routes/api");
const port = 3005;

const app = express();
app.use(cors());
app.use(express.static(path.join(__dirname, "dist")));

app.use(bodyParser.json());

app.use("/server", api);

app.listen(port, function() {
    console.log("Server running on localhost:" + port);
});
```

Here we can have the basic functionality for the server using the endpoint /server. We then need to create our API and Models for the **user** and **images**. Lets start by creating our Mongoose Schemas to properly handle the data and its types in a new folder **models/** :

User.js:

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema;
```

```
const userSchema = new Schema({
  email: String,
  password: String,
});

module.exports = mongoose.model("user", userSchema, "users");
```

image.js:

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const imageSchema = new Schema({
    title: String,
    dataurl: String,
});

module.exports = mongoose.model("image", imageSchema, "images");
```

These two schemas will allow us to properly store the post requests and retrieve the get request from the user. Mongoose allows for a quick setup of schemas which is why I opted to use mongoose along with MongDB for the database provider.

Next we need to create the API. Assume you already have a MongoDB cluster ready we will skip setting up the MongoDB Atlas database. Instead we can create a few routes and our verifyToken function:

Api.js

```
const express = require("express");
const { check, validationResult } = require("express-validator");
const router = express.Router();
const mongoose = require("mongoose");
const User = require("../models/user");
const Image = require("../models/image");
const jwt = require("jsonwebtoken");
const db =
    "yourMongoDBURL";

mongoose.connect(db, function(err) {
    if (err) {
        console.error("Error! " + err);
    } else {
```

```
        console.log("Connected to mongodb");
    }
});

function verifyToken(req, res, next) {
    if (!req.headers.authorization) {
        return res.status(401).send("Unauthorized request");
    }
    let token = req.headers.authorization.split(" ")[1];
    if (token === "null") {
        return res.status(401).send("Unauthorized request");
    }
    let payload = jwt.verify(token, "secretKey");
    if (!payload) {
        return res.status(401).send("Unauthorized request");
    }
    req.userId = payload.subject;
    next();
}

router.get("/gallery", verifyToken, (req, res) => {
    // GET LOCAL DATA DB
    let coll = mongoose.connection;
    Image.find({}, (err, result) => {
        if (err) {
            console.log(err);
        } else {
            res.send(result);
        }
    });
});

router.post("/gallery", verifyToken, (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    let cameraSnapshot = req.body;
    let image = new Image(cameraSnapshot);
    // POST DATA & LOCAL DB
```

```
    image.save((err, image) => {
        if (err) {
            console.log(err);
        } else {
            let payload = {
                subject: image._id,
            };
            let token = jwt.sign(payload, "secretKey");
            res.status(200).send({ token });
        }
    });
});

router.post("/register", [check("email").isEmail()], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    let userData = req.body;
    User.findOne({ email: userData.email }, function(err, user) {
        if (err) {
            console.log(err);
        }
        if (user) {
            return res.status(409).json({ errors: "User Already Exists!" });
        } else {
            let user = new User(userData);
            user.save((err, registeredUser) => {
                if (err) {
                    console.log(err);
                } else {
                    let payload = { subject: registeredUser._id };
                    let token = jwt.sign(payload, "secretKey");
                    res.status(200).send({ token });
                }
            });
        }
    });
});
```

```
router.post("/login", (req, res) => {
    let userData = req.body;
    User.findOne({ email: userData.email }, (err, user) => {
        if (err) {
            console.log(err);
        } else {
            if (!user) {
                res.status(401).send("Invalid Email");
            } else if (user.password !== userData.password) {
                res.status(401).send("Invalid Password");
            } else {
                let payload = { subject: user._id };
                let token = jwt.sign(payload, "secretKey");
                res.status(200).send({ token });
            }
        }
    });
});

module.exports = router;
```

Here we have the **verifyToken** function checking the header authorization from the front-end auth.service. Here we can verify the JWT token we generated before.
Then we can using the **express.Route()** to create our routes from the front end. We can handle the get requests for **gallery** to ensure the user is logged in before being able to view the gallery. We also handle the post request for when a user wants to add an image to the gallery. Along with that post request we have a router for **register** and **login.** The Login request makes sure that the user exists in the DB while the register makes sure that the Email the user uses is not already taken to allow the user to register.

Overall the backend is less code but definitely more time consuming since most of the testing and verification happens here. You can use Postman to test the requests before coding them into the front-end as well. Once we are done with our app we can create a build by running:

```
    ng build
```

this will create a **dist/** folder with our build output for later deployment.

IV. Hosting & More

As a bonus we can use Firebase to create a cloud database and generate an image URL and host our app. Although not a requirement of the challenge we can follow

Firebase guide to create our host provider there. We won't go into too much detail as firebase can get overwhelming the main features, we can just use Firebase Hosting and Storage.

On the frontend we need to add Firebase Configuration to our **environments/environment.ts** . Once we have a project created in Firebase we can copy the configuration key given. Next we need to add the **AngularFire** module to our main **app.component.ts** under the @NgModule imports**:**

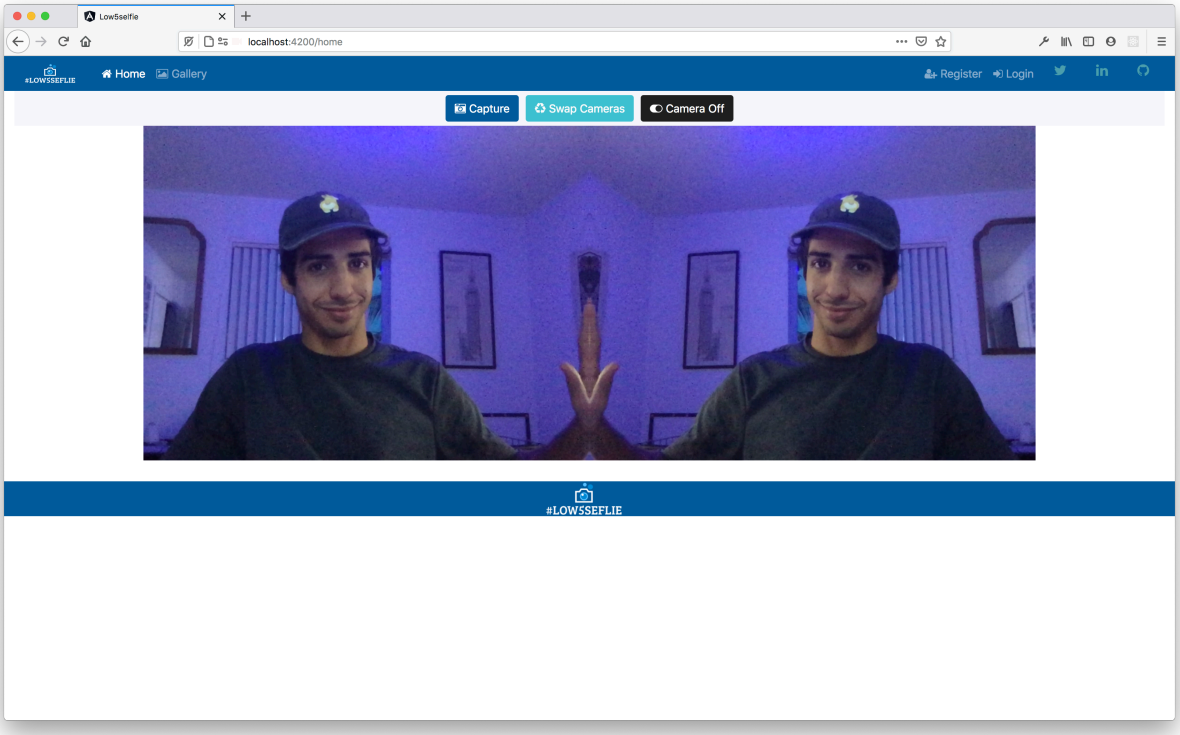
```
    AngularFireStorageModule,
    AngularFireModule.initializeApp(environment.firebaseConfig, 'cloud'),
```

Then over on the CameraViewComponent we can import Firebase and this will allow the **uploadToFirebase()** function to work properly. We have use the Firebase Tools to host our app by running the following inside **low5selfie:**

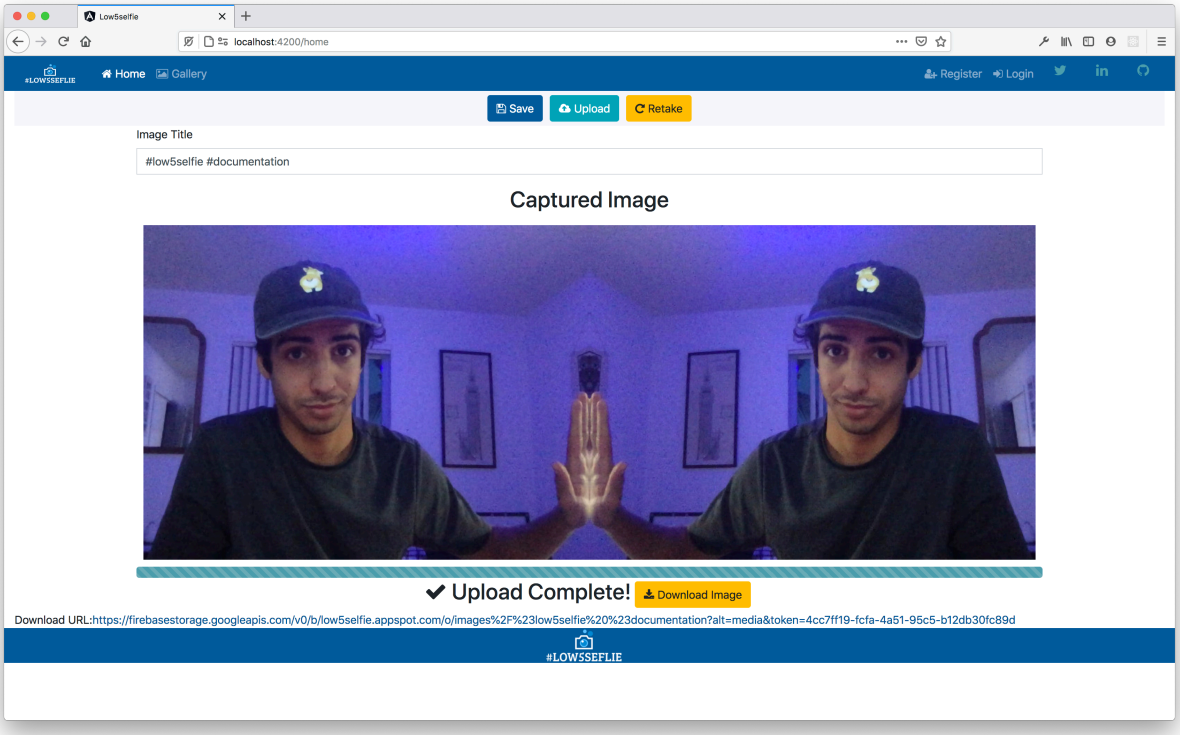
```
    firebase init
    firebase deploy
```

We want to point the deploy to the build **dist/** folder. Once deploy we can follow the app url generated by Firebase.

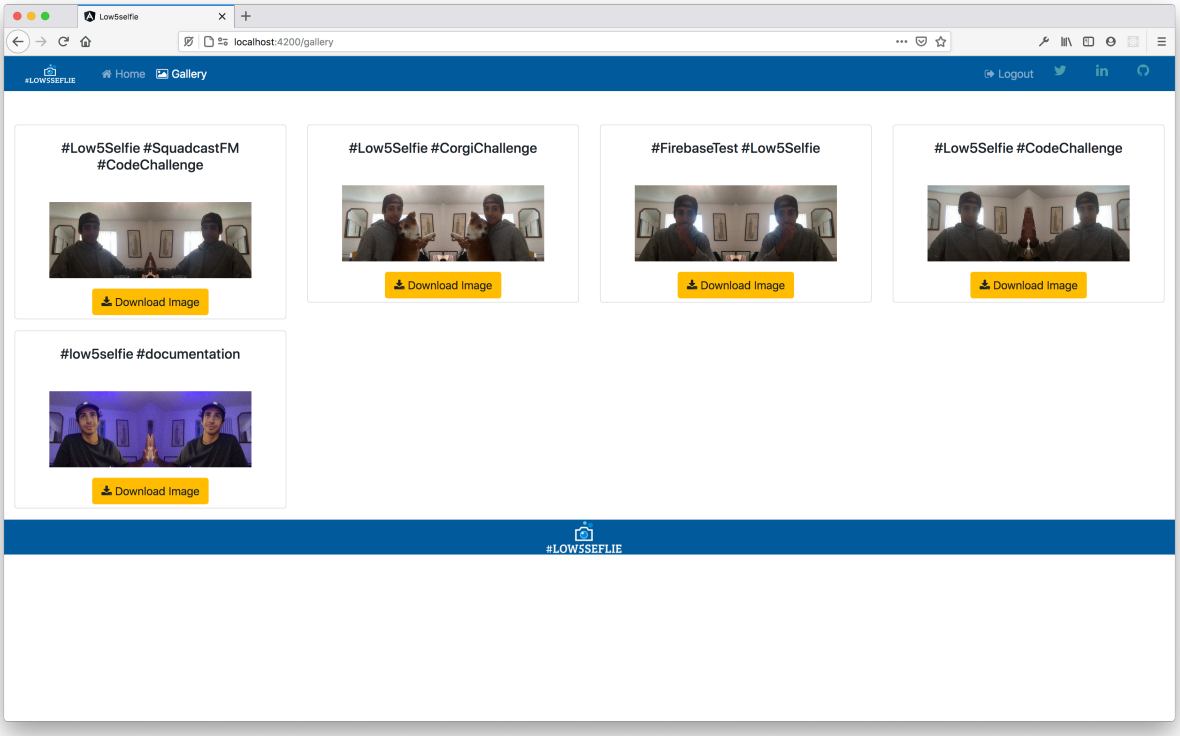
Once this is provided we can test our app! We can run the app by using npm start in both the **low5selfie** and **server** folder. Then got to **Localhost:4200** to see your progress:



We can hit **Capture** to take our low5selfie and then we can select Upload or Save to store our image.



We can then view our image in our Gallery *if you're logged in* ☺



Tada! Now we have a working Low5Selfie App to share and store all our low5s with anyone!

Challenges Breakdown

Let's break down the challenge goals:

- **Visit a single page web app**
  - **developed in Angular with TypeScript**
  - **open-source components are cool**

This part of the challenge was straight forward. I used the Angular CLI to create the boilerplate app and its components. I love angular ☺.

- **Grant permission to their camera**
  - **within the web browser**
- **View their camera's video stream**
  - **within a 2-column, side-by-side, layout**
  - **utilizing a majority of the page**
  - **styled/transformed in a way that aligns for a #Low5Selfie**

This is where my research began. I started by looking through NPM to see if there were any packages already available. I wanted to save time and just use the power of Open Source to get the main function of our app from the community. Here I choice to go with **ngx-webcam** it had everything I needed to create a sidebyside view and use **Observables** to capture and trigger the events needed to capture the image.

- **Click a button to capture the #Low5Selfie as a screenshot**
  - **use Pico, an open-source screenshot library**
  - **other open-source libraries are cool too**

This may be the only part of the challenge I couldn't complete just due to the fact that Pico had no support aside from the forums but that was a deadend. The hosting provider was also shutting down **Gripeless** I opt for my own solution to take screenshots of the image. I used the ngx-webcam capture function then used **ts-image-process** to manually alter and create a side by side image. Overall I think I spent the most time on this part of the challenge just trying to make my own work around. I also didn't want to use another screenshot package since ngx-camera already had the capture functionality.

- **Upload the #Low5Selfie image to a backend endpoint**
  - **frontend implemented as an Angular service**
  - **backend developed in Node with Express**
  - **open-source Node packages are cool**
  - **the request body should include the selfie**
  - **storing the #Low5Selfie image in the local file system is cool**
  - **the response body should include the URL of the #Low5Selfie image**

This part of the challenge I spent the most time testing and researching just trying to make sure my endpoints were fluid and used JWT tokens to verify the user. I **LOVE** using Express, Mongoose, and MongoDB for my personal projects so using it here was a breath of fresh air.

- **View the #Low5Selfie image's URL**
- **Take another #Low5Selfie**

This part I completed while setting up the other parts of the challenge. I used Firebase Storage to generate my online URL.

Bonus...

- **Switch between cameras, updating the camera stream**
  - **implemented as an observable**
- **Preview the selfie before uploading to the cloud**
- **Display selfie upload progress**
- **Secure the cloud upload endpoint with a token**
- **Validate the token with Express middleware**

For the switching between camera I have to give it once again to the ngx-camera package. I mean that package had it all to be honest and that's the main reason why I love Open Source. Previewing the selfie I coded into the HTML I used Firebase Snapshot to get the upload progress bar and secured the upload with JWT token I generated.

Super Bonus!!!

- **Present the app with a beautiful #Low5Selfie brand**
- **Deploy both apps to App Engine**
  - **no fancy domains necessary**
  - **the Google Cloud Platform trial period is free**

Alright this part I have to hand it to Firebase Hosting and a free image logo generator I found to create a small cute logo for the challenge.