# Implementation

# Frameworks/Languages

Python Javascript

Django OpenCV

## Using Django

To use Django, you need to install Python and Django.

## Create a new Django project

Once Django is installed, you can create a new project by running the following command:

django-admin startproject identity\_website

Replace project\_name with the name of your project. This command will create a new directory with the same name as your project, which will contain the basic files and folders needed to start a new Django project.

Create a new Django app: A Django project is made up of one or more apps. An app is a module that contains models, views, templates, and other code that serves a specific purpose. You can create a new app by running the following command:

python manage.py startapp identity

Replace app\_name with the name of your app. This command will create a new directory with the same name as your app, which will contain the basic files and folders needed to start a new Django app.

## The Models

Django will implement the database using the definitions in the models.py file. The developer never has to write SQL code. The models.py file is where you define the data structures for your application. Each class in models.py represents a table in the database. Each attribute of the class represents a column in the table. The following code defines a Location class that represents a table in the database.

eg.

from django.db import models  
  
  
class Location(models.Model):  
 id = models.AutoField(primary\_key=True)  
 address = models.CharField(max\_length=200)  
 description = models.CharField(max\_length=200)  
 email = models.EmailField(max\_length=200, unique=True)  
 name = models.CharField(max\_length=200, unique=True)  
 telephone = models.CharField(max\_length=20)  
  
 def \_\_str\_\_(self):  
 return self.name

creates a Table in the database

CREATE TABLE "identity\_location"   
("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,  
 "address" varchar(200) NOT NULL,  
 "description" varchar(200) NOT NULL,  
 "email" varchar(200) NOT NULL UNIQUE, "name" varchar(200) NOT NULL UNIQUE, "telephone" varchar(20) NOT NULL)

### Migrations

To create the database perform these migrations. Note 0001 increments with each migration. A second migration would be 0002 and so on.

python manage.py makemigrations identity  
python manage.py sqlmigrate identity 0001  
python manage.py migrate

python manage.py migrate

### Overriding the default User Model

In Django, user accounts are represented with a built-in User model. The Identity App requires custom fields for its User model not present in django.contrib.auth.User. To override the default User model:

1. Create a new model, in this example, named UserAccount that inherits from AbstractBaseUser classes. This new model will be used instead django.contrib.auth.User.
2. Add any additional fields you require to the new model UserAccount.
3. When referencing the UserAccount Model, use settings.AUTH\_USER\_MODEL rather than UserAccount class Roster(models.Model): id = models.AutoField(primary\_key=True) #…# user\_account = models.ForeignKey( settings.AUTH\_USER\_MODEL, on\_delete=models.CASCADE, )
4. Then set the AUTH\_USER\_MODEL setting in the Website App settings.py file to point to your new User model UserAccount with its namespace. AUTH\_USER\_MODEL = ‘identity.UserAccount’

## The Views

views.py is where you define the functions used when a user visits a particular URL. Each function in views.py acts as a controller in a traditional MVC application by connected the model with the views for your application. Views can use templates to render the HTML that is returned to the user (c.f. [The Templates](#the-templates)). Views can also return JSON data to be used by a JavaScript application.

### The Templates

Html files are stored in a project folder called templates. The templates folder must be located in the same directory as the views.py file.

For example, the following code defines a function called index that returns a rendered template website/index.html relative to the templates folder.

def index(request):  
 return render(request, 'website/index.html')

### The Static Files

Static files are files that are not generated by the application. For example, images, CSS files, and JavaScript files are static files. Static files are stored in a project folder called static. The static folder must be located in the same directory as the views.py file. To reference a static file in a template, use the static tag.

{% load static %}  
<!--...-->  
<img src="{% static 'img/identity-logo.png' %}" alt="Identity Logo"/>  
<!--...-->

### Passing Data to Templates (The Context)

The context is a dictionary that contains the data that is passed to the template. The context is passed to the render function as the third parameter.

The named parameters in the context are then available in the template as variables. For example, the following code defines a function called setup\_facial\_recognition that returns a rendered template from user-accounts/setup-facial-recognition.html relative to the templates folder.

views.py

def setup\_facial\_recognition(request):  
 user\_account = get\_object\_or\_404(UserAccount, pk=request.user.id)  
 context = { 'user\_account\_id': user\_account.id }  
 return render(request, 'user-accounts/setup-facial-recognition.html', context)

*user-accounts/setup-facial-recognition.html*

{% extends "master.html" %}  
{% load static %}  
{% block content %}  
 <div class="column">  
 <h2>User Facial Recognition Setup</h2>  
 <p>Please look directly at the camera until your face is detected and profile recorded.</p>  
 <video id="videoInput" width="320" height="240">  
 </video>  
 <p id="errorMessage" class="error"></p>  
 <input type="hidden"  
 id="user-account-id"  
 name="user-account-id"  
 value="{{ user\_account\_id }}"/>  
 <button id="startButton"  
 class="btn btn-primary"  
 onClick="setupUserFacialRecognition()">  
 Setup User Facial Recognition  
 </button>  
 <p id="statusMessage"></p>  
 </div>  
{% endblock content %}  
{% block scripts %}  
 <script src="{% static "js/facial-login.js" %}"></script>  
 <script type="text/javascript">startCamera(VideoResolutionFormatNames.QVGA, 'videoInput');</script>  
{% endblock scripts %}

### Extending templates

A base template is a template that other templates extend. Generally they contain the HTML that is common to all pages in your application with named blocks that other templates can override with view specific content. This allows you to avoid duplicating the same HTML in multiple templates. For example, a base template might contain the HTML for the header, footer, and navigation bar. Then other templates can then extend the base template and add their own HTML content for the named blocks.

For example, the following code blocks defines a base template called master.html with a title block and login.html extends the base template and overrides the title block.

master.html

<html lang="en">  
 <head>  
 <title>Identity  
 {% block title %}  
 {% endblock title %}  
 </title>  
 </head>  
 <body></body>  
</html>

login.html

{% extends 'identity/master.html' %}  
{% block title %}Login{% endblock title %}

Only the content between the block tags is needed to create view. The rest of the HTML is inherited from the base template.

### Handling 404 Errors

The get\_object\_or\_404 function is used to return a 404 error if the object is not found. For example, the following code defines a function called user\_account that returns a rendered template from user-accounts/user-account.html relative to the templates folder. If the user account is not found, a 404 error is returned.

def location\_details(request, location\_id):  
 location = get\_object\_or\_404(Location, pk=location\_id)  
 # ...

### Securing the Views

Security for an app is assisted by decorators. The @login\_required decorator is used to ensure that a user is logged in before a function is called. The @permission\_required decorator is used to ensure that a user has a particular permission before a function is called. The @csrf\_exempt decorator is used to allow a function to be called from a POST request.

If a user is not logged in, the @login\_required decorator will redirect the user to the login page. If a user does not have the required permission, the @permission\_required decorator will return a 403 error and appropriately redirect the user.

from django.contrib.auth.decorators import login\_required, permission\_required  
  
@login\_required  
@permission\_required('identity.activate\_sign\_in', raise\_exception=True)  
def sign\_in(request, location\_id):  
 location = get\_object\_or\_404(Location, pk=location\_id)  
 return render(request, 'user-accounts/sign-in-new.html', {'location': location})

### Securing blocks of the Templates

To secure blocks of the templates, use the if tag. The if tag with user.is\_authenticated function can be used to check if a user is authenticated. In the example below The if tag is used to display the appropriate navigation bar links if a user is authenticated. If a user is not authenticated, the login link is displayed.

{% if user.is\_authenticated %}  
 <li class="nav-item"><a class="nav-link" href="{% url 'locations' %}">Locations</a></li>  
 <!--...-->  
 <li class="nav-item"><a class="nav-link" href="{% url 'logout' %}">Logout</a></li>  
{% else %}  
 <li class="nav-item"><a class="nav-link" href="{% url 'login' %}">Login</a></li>  
{% endif %}

The if tag with perms.identity.activate\_sign\_in function can be used to check if a user has a particular permission. In the example below, the if tag is used to display the Activate Sign in button if the user has the identity.activate\_sign\_in permission.

<td><p id="my-desc">Actions</p></td>  
<td>  
 {% if perms.identity.activate\_sign\_in %}  
 <a class="btn btn-primary" href="{% url 'sign\_in' location.id %}">Activate Sign in</a>  
 {% endif %}  
</td>

## The Urls

In the identity project, the urls.py file is located in the identity folder. The urls.py file maps the URL to the functions defined in views.py. The path function is used to map the URL to the function. The name parameter is used to reference the URL in templates.  
Take the following example:

from django.urls import path  
from .import views  
from .views import UserEditView, PasswordsChangeView  
  
urlpatterns = [  
 # ...  
 path('', views.index, name='index'),  
 path('edit-user-profile/', UserEditView.as\_view(), name="edit-user-profile"),  
 path('locations/', views.locations, name='locations'),  
 path('locations/<int:location\_id>/',views.location\_details, name='details'),  
 # ...  
]

path('locations/', views.locations, name='locations'),

The URL is then accessed as follows: <http://localhost:8000/locations>

URl parameters are defined using angle brackets e.g. <int:location\_id>. The int parameter defines the type of the parameter. The location\_id parameter is the name of the parameter. The parameter is then available in the function as a parameter e.g. def location\_details(request, location\_id):. The URL is then defined as follows:

path('locations/<int:location\_id>/',views.location\_details, name='details')

The URL is then accessed as follows: <http://localhost:8000/locations/1/>

Generic views are used to map the URL to the function by using the as\_view function e.g. path('edit-user-profile/', UserEditView.as\_view(), name="edit-user-profile"),

The URL is then accessed as follows: <http://localhost:8000/edit-user-profile>

### Referencing the URLs

Referencing the URL in templates is done using the url function e.g.

master.html

<a class="nav-link" href="{% url 'locations' %}">Locations</a>

### Adding the project URLs to the Website

To add the URLs from the identity app to the website app, the following code path('', include('identity.urls')) is added to the website project urls.py file. '' indicates that the URLs are added to the root of the website.

from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('', include('identity.urls')),  
]

## The Admin

The admin site is a default Django application that provides a simple interface to manage content on the site. The admin site is located at /admin. The admin site is used to manage the database tables and the users.

admin.py

Custom Forms are used to add additional fields to the admin site. The custom forms are defined in the admin.py file. The custom forms are used in the following files:

## The Forms

Not to be confused with HTML forms, Django forms are used to validate the data entered by the user. The forms are defined in the forms.py file.

Crispy Forms is a third-party Django package that allows developers to create beautiful forms with less code. It provides a simpler way to layout the HTML forms using template packs, which can be easily customized according to project requirements.

The package django-crispy-forms is required to use the crispy forms in Django. The package can be installed using the following command:

pip install django-crispy-forms

In the settings.py file, the crispy forms are added to the installed apps.

INSTALLED\_APPS = [  
 # ...  
 'crispy\_forms',  
]

In the settings.py file, the crispy forms template pack is added to the template context processors.

TEMPLATES = [  
 {  
 'BACKEND': 'django.template.backends.django.DjangoTemplates',  
 'DIRS': [],  
 'APP\_DIRS': True,  
 'OPTIONS': {  
 'context\_processors': [  
 # ...  
 'django.template.context\_processors.request',  
 ],  
 },  
 },  
]

Example use of the crispy forms in the template [edit-user-profile.html](../identity-django/identity/templates/user-accounts/edit-user-profile.html)

{% extends "master.html" %}  
{% load static %}  
{% block title %}  
 Edit User Profile  
{% endblock title %}  
{% block content %}  
 <form method="post">  
 <div class="container" style="width: 30rem;">  
 <h1>Edit User Profile</h1>  
 <div class="form-group">  
 {% csrf\_token %}  
 {{ form.as\_p }}  
 <br/>  
 <button class="btn btn-primary" type="submit">Update Profile</button>  
 </div>  
 </div>  
 </form>  
{% endblock content %}

{{ form.as\_p }} is used to render the crispy form.

## Utilities

I placed the Application Code for Image Processing in utilities.py

### Constants and File Paths

Constants for FilePaths and application Settings are defined at the top of the utilities.py file.

PARENT\_DIRECTORY: Path = Path(\_\_file\_\_).resolve().parent.parent  
CLASSIFIER\_CONFIGURATION: str = str(  
 PARENT\_DIRECTORY / 'haarcascade\_frontalface\_default.xml')  
DATABASE\_DIRECTORY: str = str(PARENT\_DIRECTORY / 'database/')  
DATABASE\_FACE\_DIRECTORY: str = str(  
 PARENT\_DIRECTORY / 'database/identity\_face\_dataset')  
DATABASE\_LOG\_DIRECTORY: str = str(  
 PARENT\_DIRECTORY / 'database/log')  
DATABASE\_FACIAL\_TRAINER: str = str(PARENT\_DIRECTORY / 'database/trainer.yml')  
FACE\_CONFIDENCE\_LEVEL:float = 80.0

FACE\_CONFIDENCE\_LEVEL is the maximum confidence level acceptable that a face can be considered a match. The lower the value the more confident the application is that the face is a identified correctly. File paths are defined relative to the utilities.py file for teh directories that are used by the application, such as the database directory, the database/identity\_face\_dataset directory, and the database/trainer.yml file.

### Face Detection

The [OpenCv](https://opencv.org/) library is used to detect faces in images and to train the application to identify faces.

haarcascades are pre-trained classifiers that are used to detect objects in images. The identity project used haarcascade\_frontalface\_default.xml to detect faces in images available at [github.com/kipr/opencv](https://github.com/kipr/opencv/tree/master/data/haarcascades).

### How to use Open CV

The opencv-python package is a Python wrapper for the OpenCV library. This package must be installed to use opencv in the application. c.f. 6.4 To install and Run the project.

Import the package using the following code in python files.

import cv2

### How to find a face in an image

Given a gray-scale image Mat object as input the function detect\_user\_face returns a tuple containing a boolean flag and a numpy array. The flag is True if a face is detected in the image and False if no face is detected. The numpy array contains the coordinates of the face in the image.

[detect\_user\_face](https://github.com/Zsolt821201/iDentity-ml-fyp-py/blob/25b7eaa48465b378fb8faf3ed1d705b32a05e284/identity-django/identity/utilities.py)

def detect\_user\_face(gray\_scale\_image, min\_size=None) -> tuple[bool,ndarray]:  
 face\_detector\_classifier = cv2.CascadeClassifier(CLASSIFIER\_CONFIGURATION)  
 faces: ndarray = face\_detector\_classifier.detectMultiScale(  
 gray\_scale\_image, scaleFactor=1.3, minNeighbors=5, minSize=min\_size)  
  
 if len(faces) == 0:  
 print("Error: No face detected")  
 return False, None  
  
 if len(faces) > 1:  
 print("Error: More than one face detected")  
 return False, None  
  
 return True, faces[0]

face\_detector\_classifier = cv2.CascadeClassifier(CLASSIFIER\_CONFIGURATION) loads the classifier configuration file into the face\_detector\_classifier object. The detectMultiScale method of the face\_detector\_classifier object is used to detect faces in the image. The detectMultiScale method returns a numpy array of faces. The detect\_user\_face function checks for the cases of (1) no face detected and (2) more than one face detected. If either of these cases occur, the function returns False and None. If a single face is detected, the function returns True and the coordinates of the face in the image.

detect\_and\_save\_user\_face

def detect\_and\_save\_user\_face(user\_account\_id, image, image\_number):  
 """Detects a single face in the image and saves it to the database"""  
 gray\_scale\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)  
  
 face\_found, face = detect\_user\_face(gray\_scale\_image)  
 if face\_found:  
 x, y, w, h = face  
 directory: str = f"{DATABASE\_FACE\_DIRECTORY}/user-{user\_account\_id}"  
 os.makedirs(directory, exist\_ok=True)  
 cv2.imwrite(f"{directory}/{image\_number}.jpg",  
 gray\_scale\_image[y:y+h, x:x+w])  
  
 return face\_found

The detect\_and\_save\_user\_face function takes a user account id, an image, and an image number as input. The function converts the image to a gray-scale image and calls the detect\_user\_face function to detect a face in the image. If a face is detected, the function saves the face to the database directory. The function returns a boolean flag indicating whether a face was detected.

### How to train the model

The face\_training function trains the model using the images in the DATABASE\_FACE\_DIRECTORY directory. The function saves the trained model to the DATABASE\_FACIAL\_TRAINER file. recognizer = cv2.face.LBPHFaceRecognizer\_create() creates a face recognizer object. The recognizer.train method takes the list of images and the list of user ids as input and trains the model. With the model trained recognizer.write method saves the trained model to the DATABASE\_FACIAL\_TRAINER file.

def face\_training():  
 faces, face\_ids = get\_images\_and\_labels(DATABASE\_FACE\_DIRECTORY)  
  
 recognizer = cv2.face.LBPHFaceRecognizer\_create()  
 recognizer.train(faces, numpy.array(face\_ids))  
  
 # Ensure the database directory exists before saving the model  
 os.makedirs(DATABASE\_DIRECTORY, exist\_ok=True)  
 # Save the model into trainer/trainer.yml  
 recognizer.write(DATABASE\_FACIAL\_TRAINER)

The function get\_images\_and\_labels returns a tuple of lists (1) a list of images of the users faces from the directory DATABASE\_FACE\_DIRECTORY and (2) a list of user ids. The structure of the directory is users ids as subdirectories that contain 30 images of the users face

def get\_images\_and\_labels(path) -> tuple[list, list]:  
 face\_ids: list[str] = []  
 face\_samples: list[ndarray] = []  
  
 user\_directories = [os.path.join(path, directory)  
 for directory in os.listdir(path)]  
 for user\_directory in user\_directories:  
 user\_image\_files = [os.path.join(  
 user\_directory, file) for file in os.listdir(user\_directory)]  
 face\_id = int(os.path.split(user\_directory)[-1].split("-")[1])  
  
 for image\_path in user\_image\_files:  
 face\_image\_numpy: ndarray = numpy.array(  
 Image.open(image\_path), 'uint8')  
 face\_samples.append(face\_image\_numpy)  
 face\_ids.append(face\_id)  
  
 return face\_samples, face\_ids

### How to recognize a face in an image

To recognize a face in an image, you need a trained model i.e.[trainer.yml](../identity-django/database/trainer.yml) . The face\_recognition\_web function takes an image as an input and calls the face\_image\_recognition function to recognize a face in the image.

def face\_recognition\_web(open\_cv\_image: ndarray):  
 recognizer:cv2.face.LBPHFaceRecognizer = cv2.face.LBPHFaceRecognizer\_create()  
 recognizer.read(DATABASE\_FACIAL\_TRAINER)  
  
 user\_id, confidence, face = face\_image\_recognition(recognizer, open\_cv\_image)  
  
 return user\_id, confidence

The face\_image\_recognition function takes a recognizer object, an image, and a minimum size as input. The function converts the image to a gray-scale image and calls the [detect\_user\_face](#how-to-find-a-face-in-an-image) function to detect a face in the image. If a face is detected, the function calls the recognizer.predict method to recognize the face. The recognizer.predict method returns a tuple containing the user id and the confidence of the prediction. The face\_image\_recognition function then returns a tuple containing the user id, the confidence of the prediction, and the coordinates of the face in the image.

def face\_image\_recognition(recognizer: cv2.face.LBPHFaceRecognizer, image, min\_size = None):  
 gray\_scale\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)  
  
 is\_face\_present, face = detect\_user\_face(gray\_scale\_image, min\_size=min\_size)  
   
 if(not is\_face\_present):  
 return None, 0, None  
   
 x, y, width, height = face  
 user\_id, confidence = recognizer.predict(gray\_scale\_image[y:y+height, x:x+width])  
   
 return user\_id, confidence, face

### File Encoding

To work with images in Python, you need to convert the image to a numpy array. The image\_to\_numpy\_array function takes an image file path as input and returns a numpy array. The function uses the PIL.Image.open method to open the image file and the numpy.array method to convert the image to a numpy array.

However, images sent via HTTP requests are encoded in Base64. To work with OpenCV these images need to be decoded. The decode\_base64 function takes a Base64 encoded string as input and returns a byte array. The function uses the re.sub method to remove the data:image/.+;base64, prefix from the string. The base64.b64decode method then decodes the string.

import base64  
import re # Regular expression  
  
def decode\_base64(image\_base64\_str):  
 image\_data = re.sub('^data:image/.+;base64,', '', image\_base64\_str)  
 return base64.b64decode(image\_data)

The stream\_image function takes a Base64 encoded string as input and returns a BytesIO object. The function uses the decode\_base64 function to decode the string and the BytesIO method to create a BytesIO object.

from io import BytesIO  
  
def stream\_image(image\_base64\_str: str) -> BytesIO:  
 return BytesIO(decode\_base64(image\_base64\_str))

This BytesIO object can be passed to the numpy.array and PIL.Image.open methods to create a numpy array that can be used with the OpenCV functions.

import numpy  
from PIL import Image  
  
def parse\_roaster\_signing\_requests(request) -> tuple[bool, UserAccount]:  
 request\_data = request.POST  
 image\_bytes = stream\_image(request\_data['image-base64'])  
 open\_cv\_image = numpy.array(Image.open(image\_bytes))  
 #...

### Creating Sample users

For demonstration purposes, sample users were created using Youtube videos. The following code creates a sample user taking a video from Youtube.

def build\_sample\_user(video\_path:str, session\_user\_account\_id:str):  
   
 video\_capture = cv2.VideoCapture(video\_path)  
 face\_detector\_classifier = cv2.CascadeClassifier(CLASSIFIER\_CONFIGURATION)  
  
 image\_number: int = 0  
 FACE\_SAMPLE\_COUNT=30  
 ESCAPE\_KEY: int = 27  
 while(image\_number < FACE\_SAMPLE\_COUNT):  
 is\_video\_capture\_open, open\_cv\_image = video\_capture.read()  
  
 if not is\_video\_capture\_open:  
 print("Error: Camera is not opened")  
 break  
  
 cv2.imshow('image', open\_cv\_image)  
  
 gray\_scale\_image = cv2.cvtColor(open\_cv\_image, cv2.COLOR\_BGR2GRAY)  
  
 faces: ndarray = face\_detector\_classifier.detectMultiScale(  
 gray\_scale\_image, 1.3, 5)  
  
 face\_found = detect\_and\_save\_user\_face(  
 session\_user\_account\_id, open\_cv\_image, image\_number)  
 if face\_found:  
 image\_number += 1  
  
   
  
 for (x, y, w, h) in faces:  
 cv2.rectangle(open\_cv\_image, (x, y), (x+w, y+h), (255, 0, 0), 2)  
 cv2.imshow('image', open\_cv\_image)  
  
 if cv2.waitKey(1) == ESCAPE\_KEY:  
 break  
 # Do a bit of cleanup  
 video\_capture.release()  
 cv2.destroyAllWindows()

## The Website

The website app is used to display the content on the website. The website app is located in the website folder [identity\_website](../identity-django/identity_website).

### Linking the App to the Website

To link the app to the website, add the app to the INSTALLED\_APPS list in the settings.py file.

INSTALLED\_APPS = [  
 'identity.apps.IdentityConfig',  
]

### Configuring Login Paths

In the settings.py to set the login URL and the login redirect URL, add the following code.

LOGIN\_URL = '/login/'  
LOGIN\_REDIRECT\_URL = '/login/'

## Client Side Functionality

To send a video stream from the client machine to the server, Javascript is used.

There are three use cases that server listens for

* User facial recognition Setup (Use case #1)
* Location Roaster Sign in (Use case #2)
* Location Roaster Sign out (Use case #3)

### Use Case #1: User Facial Recognition Setup

the first code to developed was how can a user setup their facial recognition.

The User Facial Recognition Setup was implemented as a set of procedural functions.

Development/test of the User Facial Recognition Setup highlighted difficulties with the use of procedural functions. The procedural functions were difficult to test and difficult to maintain. It was generally difficult to reuse the code for the other use cases.

### Location Roaster Sign in/out

Both use cases 2 and 3 are similar. They both send a video stream to the server, a location id, and a user id to teh server The only difference between the use cases 2 and 3 is the action the server takes. Use case 2 is a sign in and use case 3 is a sign out. In the client Javascript code, the action to either sign in or sign out is decided by the URL that the request is sent to.

Given the issues with the procedural functions for use case 1, the client side Javascript code was refactored to use classes. The classes were designed to be reusable for use cases 2 and 3 and independent of the html page elements. This was accomplished by use of dependency injection. The html page elements are passed to the class as parameters during construction.

1. const webPageControls = new WebPageControls()

const manager = new Manager(webPageControls, UrlPaths.PERFORM\_SIGN\_OUT\_URL);

2.

WebPageControls is a class that contains the html page elements. The class is used to abstract the html page elements from the rest of the code. This allows the code to be reused for different html pages. The class also provides a single point of access to the html page elements. This allows the code to be more maintainable.

Manager is a class that is used to manage the video stream. The class is used to abstract the video stream from the rest of the code. This allows the code to be reused for different use cases. The class also provides a single point of access to the video stream. This allows the code to be more maintainable. Manager is ‘dependent’ on the WebPageControls class. The WebPageControls class is passed to the Manager class as a parameter ‘injected’ during construction. Hence the term dependency injection.

The second parameter to the Manager class is the URL that the video stream is sent to. This allows the Manager class to be reused for use cases 2 and 3 by passing in the appropriate URL.

[facial-login.js](../identity-django/identity/static/js/facial-login.js)

### Sending a Video Stream

### Receiving a Video Stream

### Displaying a Video Stream

### Using Javascript

Events should be attached to HTML elements with javascript rather than in the HTML.

Issues with Javascript include the page loading before the Javascript is loaded.

To ensure that Javascript is not run before the page is loaded, the following code is used.

$(document).ready(function() {  
 //Your code goes here  
});

### Third Party Libraries

* [JQuery](https://jquery.com/)
* [Bootstrap](https://getbootstrap.com/)