

氏名：成川 喬朗
学籍番号：101830282

概要

今回は Python WSGI を用いて Web アプリケーションを作成した。

このアプリケーションでは人の顔写真が 2 つ表示されどちらかを選択するのだが、片方は実在する人の写真であり、片方は GAN（Generative Adversarial Network）というディープラーニングの技術を用いて生成されたものである。そして、どちらが本物の実在する人の顔写真であるかを人間に判断させるというものである。

図 1 は本物と偽物の顔写真が並べて表示されており、本物の顔写真はデータセット Flickr-Faces-HQ Dataset [1] に含まれるもの、偽物の顔写真は 2020 年現在最先端とされている StyleGAN2 [2] によって生成されたものである。今回はこの 2 種類の画像を使用してアプリケーションを作成した。

このアプリケーションは Facebook の CEO マーク・ザッカーバーグが大学生時代に作成した Facemash [3] の影響を強く受けており、肝となっている 2 つの人の顔写真からどちらかを選ぶ部分は特に参考になっている。



図 1. 本物と生成された画像

アプリケーションについて

図 1 は 2 つの写真から本物の顔写真を選ぶ画面である。画像下に配置されているボタンを押すことで写真を選択することができる。また、表示されているそれぞれの画像は毎回ランダムに選択されるので、チャレンジする度に異なる問題に挑戦することができる。

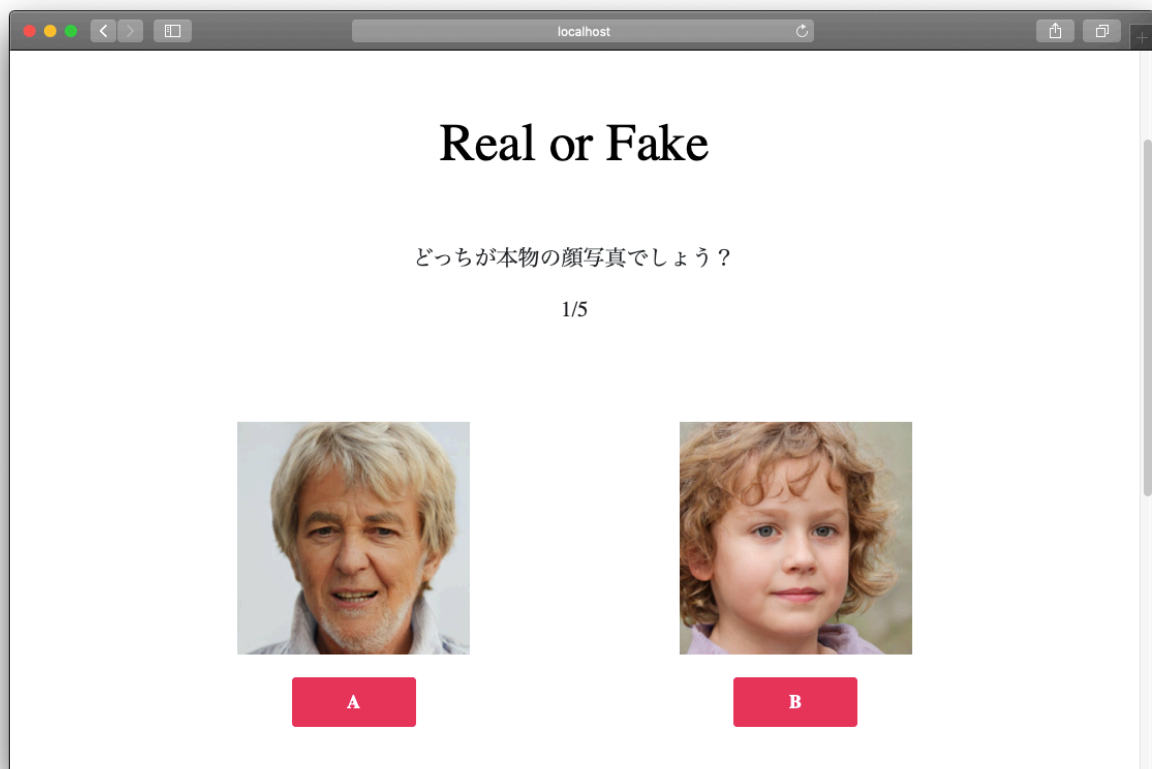


図 2. 本物と偽物の選択画面

全ての問題を完了すると図 2 のような画面が表示され、結果を確認することができる。赤枠で囲まれている方がユーザーが選択した画像であり、画像の下にマルが表示されていれば本物の画像、バツが表示されていれば偽物の画像である。画面上部にはユーザーの正答率を確認することができる。



図 3. 結果画面

このアプリケーションのトップページでは図 3 のような画面があり、生成された偽物の画像のリーダーボードが確認できる。ここで上位に表示されているほど、より人を騙した本物と見分けがつかない生成画像ということである。

画像リンクをクリックすると図 4 のようなページに移動し、どんな画像であるかを確認することができる。

リーダーボード

最も人を騙した生成画像一覧

#	騙した割合	問題に使用された回数	人を騙した回数	画像リンク
1	0.60	88	53	Link
2	0.60	88	53	Link
3	0.60	93	56	Link
4	0.60	99	59	Link
5	0.59	106	63	Link
6	0.58	79	46	Link
7	0.58	105	61	Link
8	0.58	119	69	Link
9	0.57	102	58	Link
10	0.57	111	63	Link

図 4. リーダーボード

生成画像の戦績



順位	1 / 100 位
騙した割合	0.6023
問題に使用された回数	88
人を騙した回数	53

[トップページに戻る](#)

図 5. リーダーボードの詳細


利用方法

サーバー側の操作

サーバー側を起動するには、図 5 のように実行する。

ディレクトリを `ROOT/cgi-bin` に変更し、Python のプログラム `run.py` を実行する。

また、図 6 のように実行時にはオプションを使用することができ、`--port` で使用するポートを、`--dbname` で使用するデータベースのファイル名を指定することができる。




```
~/Code/Univ/Python-WSGI
> cd cgi-bin

~/Code/Univ/Python-WSGI/cgi-bin
> python3 -m run
Application initialized

Starting Python WSGI Server ...
address: http://localhost:8080
local address: http://192.168.1.2:8080/
```

図 6. サーバーを起動するコマンド



```
~/Code/Univ/Python-WSGI
> cd cgi-bin

~/Code/Univ/Python-WSGI/cgi-bin
> python3 -m run
Application initialized

Starting Python WSGI Server ...
address: http://localhost:8080
local address: http://192.168.1.2:8080/
```

図 7. オプションを用いてサーバーを起動するコマンド

ブラウザ側の操作

サーバーを実行した時に表示された `local address`、もしくは同じ PC であれば `address` のアドレスにブラウザからアクセスすることで、アプリケーションを開くことができる。少しスクロールすると、図 8 のような問題数を選択する画面が出てくるので、ボタンを押すことで問題を開始することができる。

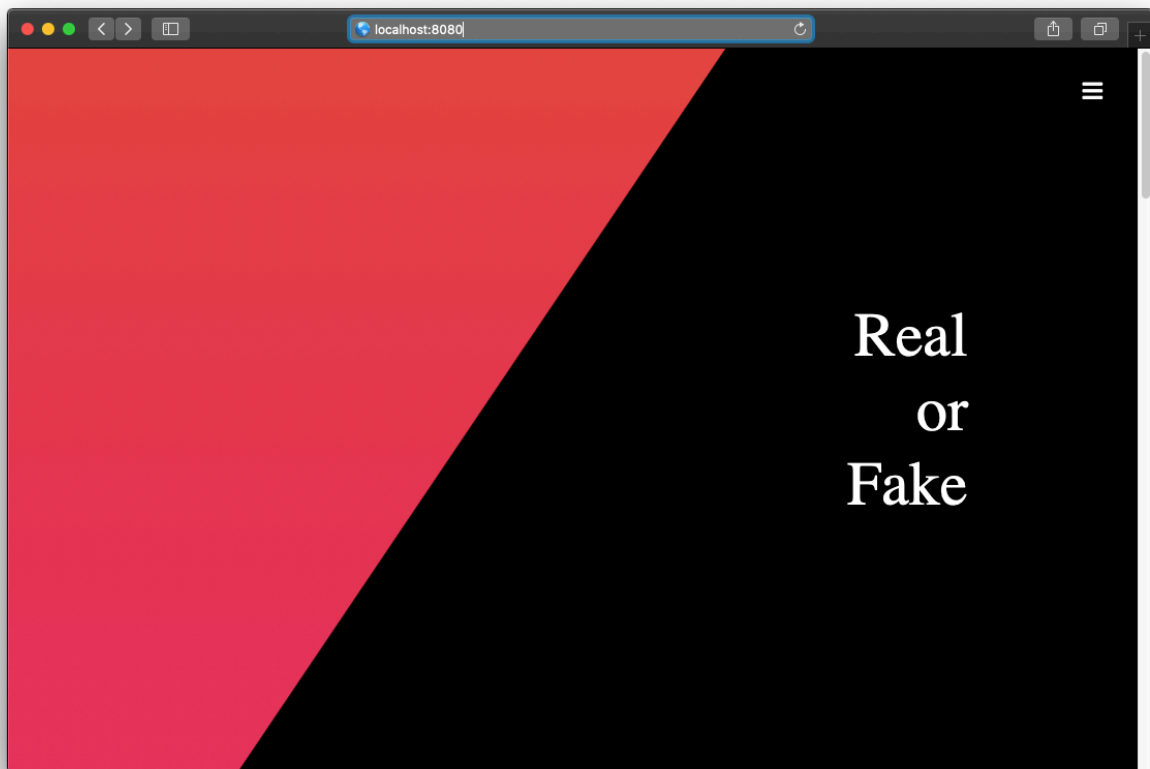


図 8. ブラウザからアクセスした様子



図 9. 問題を始める画面

作成上の工夫点

- 機能を分割し、スケーラブルな開発環境に

サンプルとして用意されていた `test_wsgi.py` では、HTML の内容を決定したりデータベースを操作したりなど、全ての機能が集約されていた。このままではページ数が増えてきたりページの機能が複雑になってきた場合開発を進めることが困難になるため、これらの機能を切り分けた。作成したアプリケーションのファイル構成は図 10 のようになった。

`cgi-bin/` 直下に配置されている `app.py` がアプリケーション表示を管理する役割を果たし、`cgi-bin/pages/` に配置される `index.py` や `bad_request.py` などでは各ページの表示内容を定義した。また HTML や CSS、画像などは `static/` に配置した。

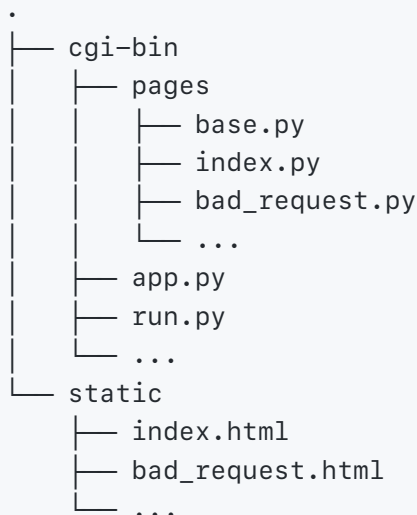


図 10. アプリケーションのファイル構成

`app.py` に定義されている `Application` クラスがアプリケーションを管理しており、ブラウザ側からリクエストを受け取ると `Application.__call__` が実行される。`Application.__call__` では、ルーティングの役割を果たす `Application.routing` 関数から `status` , `headers` , `body` の 3 つを受け取り、戻り値として `[body]` を返している。ルーティングの部分については後述する。

この役割のおかげで、`env` の状態ごとに `status` , `headers` , `body` の内容を変えることができるようになった。その結果、それらの内容を図 12 のように各ページごとに別ファイルで定義することができるようになった。

```
class Application:
    def __call__(self, env, start_response):
        status, headers, body = self.routing(env, start_response)
        start_response(status, headers)
        return [body]
```

図 11. `Application.__call__` の内容


```

def body(self, env):
    gen_id = random_string()
    return self.load_html("../static/index.html", embedding_dict={"id":
gen_id})

def status(self):
    return '200 OK'

def header(self, content_length, env):
    return [
        ('Content-Type', 'text/html; charset=utf-8'),
        ('Content-Length', str(content_length))
    ]

```

図 12. ページクラスにおける body, status, header 関数

• HTML ファイルを別で定義することで、Python と HTML を分離

サンプルプログラムは Python 上で HTML の内容を定義していたが、このままでは処理を行う部分と HTML の部分が混同してしまうため、HTML ファイルを別で用意することにした。しかし、その場合 Python 上の変数等を埋め込む必要があるため、変数を埋め込んだ状態で HTML テキストを読み込めるような仕組みを実装した。

図 12 の `body` 関数でも使用されているが、各ページの親クラスには `load_html` という関数を実装した。これを使用することにより、`embedding_dict` に指定された内容を HTML テキストに埋め込むことができる。

具体的な使用例を図 13 に示す。`index.html` の一部に `%foo%` のように変数名が `%` で囲まれた部分を用意する。そして、Python から `load_html` で読み込む際に、`embedding_dict` に先程 `%` で囲んだ変数名に埋め込みたい内容を指定する。

```

<div>
  Lorem %foo% Ipsum
</div>

```

図 13. load_html の HTML 側の使用例

```

body = self.load_html('index.html', embedding_dict={"foo": 123})

```

図 14. load_html の Python 側の使用例

すると、Python 側に読み込まれる HTML テキストは図 15 のようになる。

これにより、Python の処理部分と HTML の部分を分割しつつ、Python の変数を簡単に埋め込むことができるようになった。

```
<div>
  Lorem 123 Ipsum
</div>
```

図 15. load_html を使用して読み込まれる内容

• ルーティング機能

このアプリケーションには複数のページが必要であり、URL によって表示するページを切り替える必要がある。そのため、`Application` クラスには `URL` と表示ページの対応付けを `self.router` に定義し、`Application.routing` で使用することでルーティング機能を実装した。

また、`self.router` のキーには正規表現を使用している。これはページにパラメータが入った場合、`/challenge?n=5&id=xerh` のようになり、様々な文字列が後に連結される可能性があるからである。正規表現をキーにして辞書から値を取り出すために、`/cgi-bin/utils.py` に定義されている `RegexDict` を使用した。

```
self.router = {
    r'.*\.css': resource.Css,
    r'.*\.js' : resource.Js,
    r'.*\.png': resource.Image,
    r'.*\.jpg': resource.Image,

    r'^/$'           : pages.Index,
    r'^/challenge.*$': pages.Challenge,
    r'^/result.*$'   : pages.Result,
    r'^/image.*$'    : pages.FakeImageRecord,
}
```

図 16. URLと表示ページの対応

また、ルーティングを自前で管理することにより、HTML から呼ばれる JavaScript や CSS、画像などのリクエストに対応する必要性が生じた。そのため、`self.router` にはページに関するものだけでなく、`.*\.css` や `.*\.png` などのテキストファイルや画像などに関するものも定義した。

図 17 は CSS のリクエストに対するものである。テキストの場合は Python 上でファイルを読み込み、`utf-8` でエンコードしたものを返すことでうまくいった。また、ヘッダ情報には CSS の場合 `Content-Type` を `text/css` にするなど、適切な値に変更する必要があった。

図 18 は 画像のリクエストに対するものである。画像の場合は `open(mode='rb')` により画像ファイルをバイナリとして読み込み、それを返すことでうまくいった。ヘッダ情報には、その画像ファイルのサイズを乗せる必要があり、Python の `os` モジュールを用いてデータサイズを調べ、それを値として乗せる必要があった。

```

def body(self, env):
    request_path = env.get("PATH_INFO")

    with open(f"..{request_path}") as f:
        static = f.read().encode('utf-8')

    return static

def header(self, content_length, env):
    return [
        ('Content-Type', 'text/css; charset=utf-8'),
        ('Content-Length', str(content_length))
    ]

```

図 17. CSSのリクエストに対する処理

```

def body(self, env):
    request_path = env.get("PATH_INFO")
    try:
        with open(f"..{request_path}", "rb") as f:
            static = f.read()

    except Exception as e:
        raise (e)

    return static

def header(self, content_length, env):
    request_path = env.get("PATH_INFO")
    ext = os.path.splitext(request_path)[1][1:]
    return [
        ('Content-Type', f'image/{ext}'),
        ('Content-Length', str(os.stat(f"..{request_path}").st_size))
    ]

```

図 18. 画像のリクエストに対する処理

- 問題中の状態保持

問題に回答するページでは、ユーザの問題への回答を保持しておく必要があるので、回答を行うごとにサーバー側のデータベースに保存することにした。また、フロントからサーバーにデータを送信する手法は、URLのパラメータを用いることにした。図 19 は URL の例である。各パラメータの役割は表 A のようになっている。

```
http://localhost:8080/challenge?n=5&i=1&id=pomvpbgdfivztdy&ans=B
```

図 19. 問題ページのURLの例

表A 問題ページのURLのパラメータ

URL 中のパラメータ	説明
n	問題数
i	回答した問題数
id	ランダムに生成されるユーザー ID
ans	前回の問題の回答

ユーザーの回答は、図 20 のように回答の選択ボタンに `ans` を含めることでサーバーに送っている。

```
<a href="challenge?n=5&i=2&id=pomvpgbgdfivztdy&ans=A"><span>A</span></a>
<a href="challenge?n=5&i=2&id=pomvpgbgdfivztdy&ans=B"><span>B</span></a>
```

図 20. 問題ページのボタン

調査課題

調査課題では Web アプリケーションのセキュリティを脅かす脆弱性について調べた。まず、Webアプリの開発環境のトレンドを調べてみると、フロントエンドでは HTML と CSS で直接書くことはなく、React.js や Vue.js など、JavaScript フレームワークを使用して開発されていることが分かった [4]。特に、React.js が最も使用されていることが分かった [5]。

今回主に調べたことは、サーバーとのやり取りにおいてログイン情報をどのように保存するかである。HTTP は前回の情報などを保持することができないプロトコルであり、通信を行う度に情報がリセットされる。そのため、このままでは例えばWebアプリにログインしたとしても、ページにアクセスし直すとログアウトしてしまう。この問題を解決するための手法は大きく分けて Cookie と Session の2つ存在する [6]。

どちらもクライアントを識別する情報をブラウザ側で保持する必要があり、いくつか手法が存在するが、JavaScript では Local Storage を使う方法がある。Local Storage は JavaScript に組み込まれた機能であり、ライブラリなどを用いずに使用することができるが、これには脆弱性が存在すると言われている [7]。

Local Storage に存在する脆弱性がどんなものかと言うと、あらゆる JavaScript コードからアクセスできてしまうことである。これはWebアプリの開発者自身が書いたコードだけでなく、ライブラリなどからも同様にアクセスできてしまう。つまり、インストールしたライブラリの中に悪意のあるものが含まれていた場合、開発したアプリに第三者がログインできる可能性が生じてしまう。

しかしながら、未だに Local Storage を使用する開発者は存在し、むしろ Local Storage でも問題ないとする人も存在する [8]。

- 実際に検証

上記の内容を実際に試してみる。まず、Local Storage の内容を盗み見るライブラリとして、Node.js で 図 21 のようなライブラリを作成した。Local Storage の `secret-info` キーに含まれる情報を読み取っている。これを GitHub にアップロードし、ライブラリとしてインストールできるようにした。

```
const info = localStorage.getItem("secret-info");
console.log(`"secret-info" in localStorage: ${info}`);
```

図 21. Local Storage を盗み見るライブラリ

次に、図 22 のような Local Storage にデータを保存する Web アプリを React.js で作成した。Local Storage の `secret-info` キーに情報を保存している。先程作成したライブラリをインストールするためには図 23 のコマンドを実行する。

```
import React from "react";
import ReactDOM from "react-dom";
import "localstorage-stealer-sample";

localStorage.setItem("secret-info", "Your Password");
ReactDOM.render(<h1>This is React App</h1>,
document.getElementById("root"));
```

図 22. Local Storage に情報を保存する Web アプリ

```
$ yarn add https://github.com/TakaoNarikawa/LocalStorage-Stealer-Sample.git
yarn add v1.22.4
warning ../package.json: No license field
[1/4] 🔍 Resolving packages...
[2/4] 🚚 Fetching packages...
[3/4] 🔗 Linking dependencies...
[4/4] 🛠 Building fresh packages...
success Saved lockfile.
success Saved 1 new dependency.
info Direct dependencies
└─ localstorage-stealer-sample@1.0.0
info All dependencies
└─ localstorage-stealer-sample@1.0.0
✨ Done in 13.11s.
```

図 23. 作成したライブラリをインストールする様子

- どのように認証を保持すべきか

上記で説明した脆弱性などは時間が経つごとに更新され、今まで安全だった手法が危険になっている可能性も十分ある。そのため、認証をセキュアに保つ一番簡単な方法は、自前でサーバーとの認証部分を構築するのではなく、信頼できるサービスやライブラリなどに認証周りを任せてしまうことである。そうすれば、自身で脆弱性に対応することなく常に Web アプリをセキュアに保つことができる。例をあげると Auth0[9] などが存在する。

引用

- [1] Tero Karras, Samuli Laine, Timo Aila (<https://arxiv.org/abs/1812.04948>) (2019 年 3 月)
- [2] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, Timo Aila - Analyzing and Improving the Image Quality of StyleGAN (<https://arxiv.org/abs/1912.04958>) (2019 年 3 月)
- [3] Poulami Nag - Mark Zuckerberg turns 33: A roll call of Facebook creator's life (<https://www.ibtimes.sg/mark-zuckerberg-turns-33-roll-call-facebook-creators-life-10060>) (2017 年 3 月)
- [4] Svitlana Varaksina and Artem Chervichnik - The Latest Trends in Web App Development for 2020: What to Expect from the Industry(<https://themindstudios.com/blog/web-app-development-trends/>) (2019 年 11 月)
- [5] The State of JavaScript 2019(<https://2019.stateofjs.com/front-end-frameworks/>) (2020 年 12 月)
- [6] Takayuki Watanabe - セッション管理の周辺知識まとめ(<https://blog.takanabe.tokyo/2014/12/%E3%82%BB%E3%83%83%E3%82%B7%E3%83%A7%E3%83%B3%E7%AE%A1%E7%90%86%E3%81%AE%E5%91%A8%E8%BE%BA%E7%9F%A5%E8%AD%98%E3%81%BE%E3%81%A8%E3%82%81>) (2014 年 12 月)
- [7] Randall Degges - Please Stop Using Local Storage (<https://www.rdegges.com/2018/please-stop-using-local-storage/>) (2018 年 1 月)
- [8] Storing Authentication Tokens - Local Storage or Cookies? (https://www.reddit.com/r/Angular2/comments/cubdwa/storing_authentication_tokens_local_storage_or/) (2019 年 8 月)
- [9] Auth0 (<https://auth0.com/jp/>)