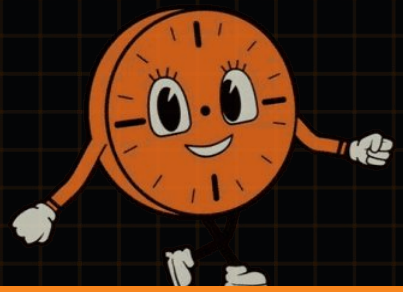
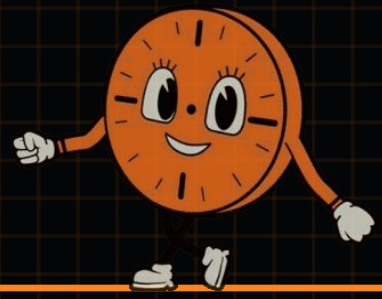
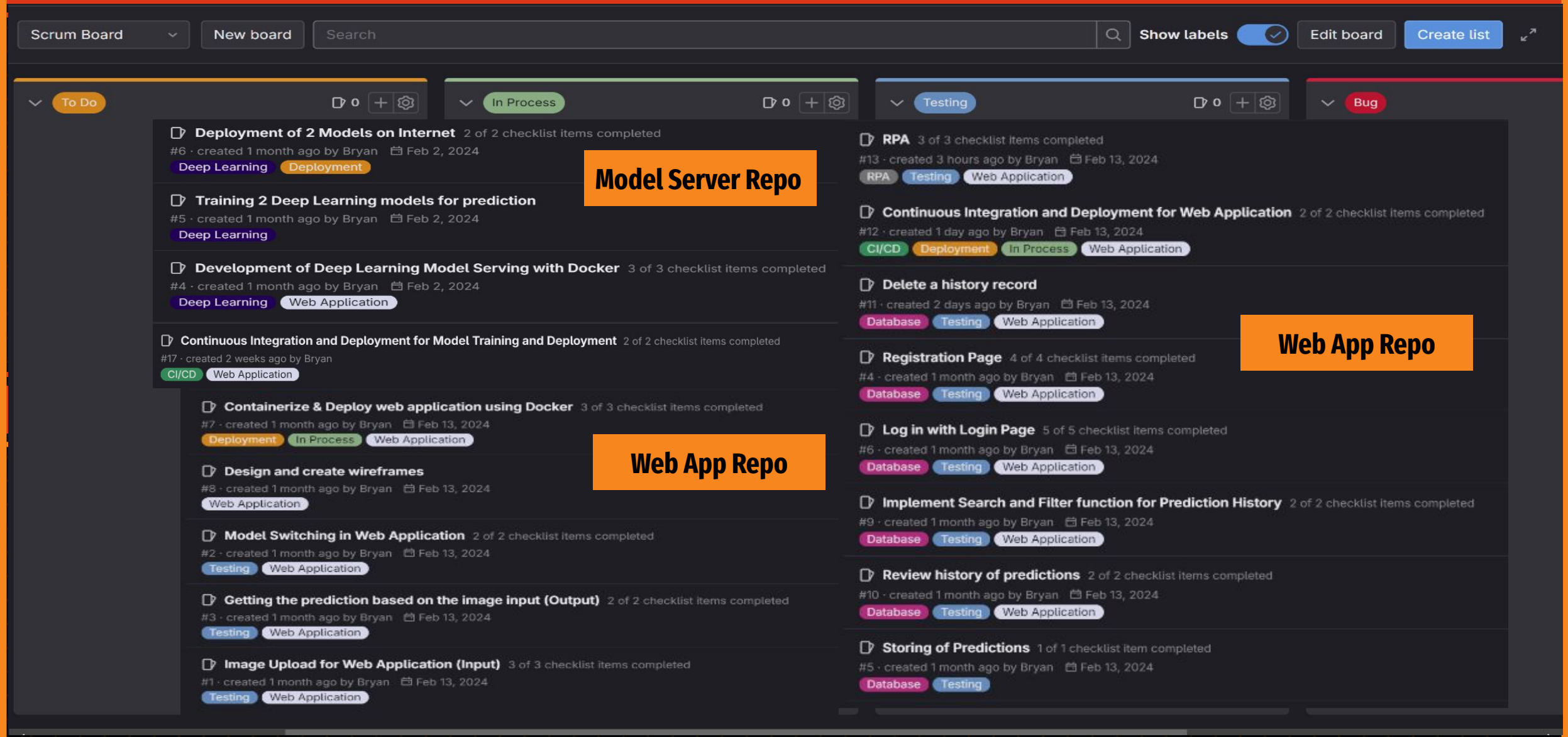


DevOps CA2

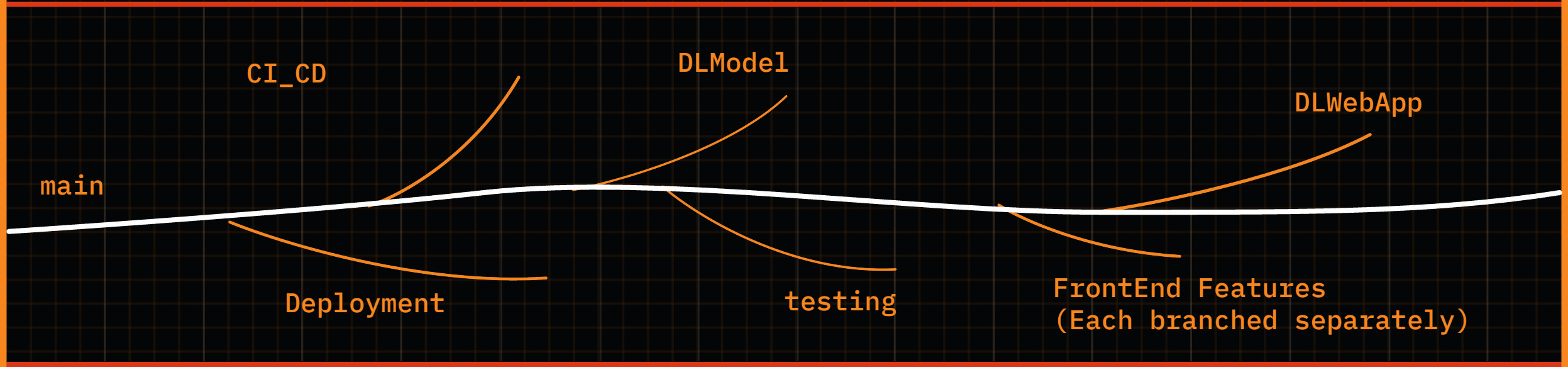
TAN WEN TAO BRYAN
DAAA/FT/2B/01



Issues & Labels



Git Branches & Commits



main default protected
cc0046f6 · Updated README · 43 minutes ago

42 Commits
3 Branches
66 Commits
11 Branches

testing
811e0dd4 · Stuff: · 57 minutes ago

DLWeb_SearchFn_App
fd462a44 · Fixed the filter by sorting of dates and rearranging search bar such that apis do not interfere · 15 hours ago
12|0

CI_CD
0d5a5a22 · Complete Deployment for CICD process · 1 day ago
14|0

Deployment
f76405a9 · Add gunicorn to requirements.txt · 1 day ago
15|0

DLWeb_History_App
91324cb0 · Fixed the feature of displaying images in the history table · 1 day ago
25|0

DLWeb_ForgotPass_App
5b1e2a78 · Ensure JWT expiration and implemented password recovery function · 2 days ag

DLWeb_Login_App
107c17c1 · Included Login Page and verification of the user credential in database · 2 days

DLWeb_Predict_App
eb601c73 · Last min fixes · 3 days ago

DLWeb_Index_App
e12df396 · Added footer · 4 days ago

DLWebApp
a1e8f18e · Added wireframes and resolved bugs · 2 weeks ago

Model Deployment

Models Deployment:

- https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_31x31
- https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_128x128

```
FROM tensorflow/serving
COPY / /
ENV MODEL_CONF=/img_classifier/models.config MODEL_BASE_PATH=/
EXPOSE 8500
EXPOSE 8501
RUN echo '#!/bin/bash \n\n\
tensorflow_model_server \
--rest_api_port=$PORT \
--model_config_file=${MODEL_CONF} \
"$@"' > /usr/bin/tf_serving_entrypoint.sh \
&& chmod +x /usr/bin/tf_serving_entrypoint.sh
```

Deploying both models locally

- Serve the 2 models on tensorflow-serving and deploy it locally

```
docker run --name vegetable_cnn -p 8501:8501 -v "D:/ca2-daaa2b01-2214449-tanwentaobryan-img_classifier/img_classifier:/models/img_classifier" -t tensorflow/serving --
model_config_file=/models/img_classifier/conf/models.config &
```

- Use of **tensorflow serving** to serve both models previously saved as tf format
- Use of **Docker** to **containerize** both models and the dependencies through a **config** file
- Deploy on Render for public access

```
model_config_list: {
  config: {
    name:  "veggie_cnn_31x31",
    base_path:  "/img_classifier/veggie_cnn_31x31",
    model_platform: "tensorflow",
    model_version_policy: {
        all: {}
    }
  },
  config: {
    name:  "veggie_cnn_128x128",
    base_path:  "/img_classifier/veggie_cnn_128x128",
    model_platform: "tensorflow",
    model_version_policy: {
        all: {}
    }
  }
}
```

Export the models

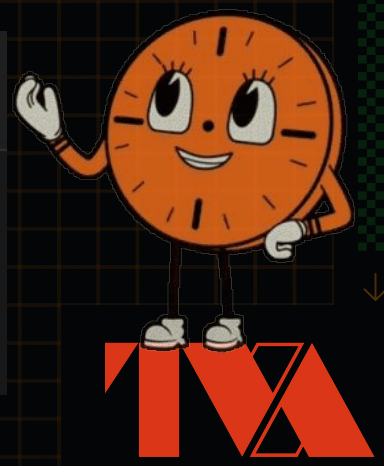
```
# Load the models weights
veggie_cnn_31x31 = load_model('./DL_models/Final_31x31_Model.h5')
veggie_cnn_128x128 = load_model('./DL_models/Final_128x128_Model.h5')

# Save the models into img_classifier folder
save_model(veggie_cnn_31x31, "./img_classifier/veggie_cnn_31x31", save_format="tf")
save_model(veggie_cnn_128x128, "./img_classifier/veggie_cnn_128x128", save_format="tf")
```



```
{
 "model_version_status": [
  {
   "version": "202401",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```

```
{
 "model_version_status": [
  {
   "version": "202402",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```


Outputs of both deployed models



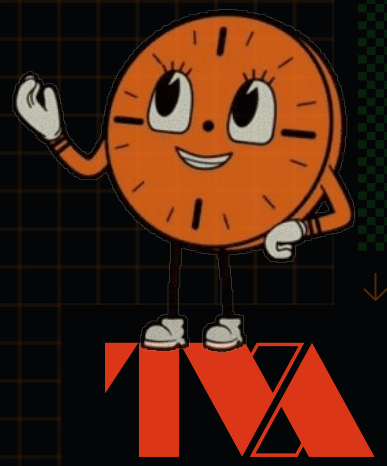
Model Testing

Testing **remotely deployed** models

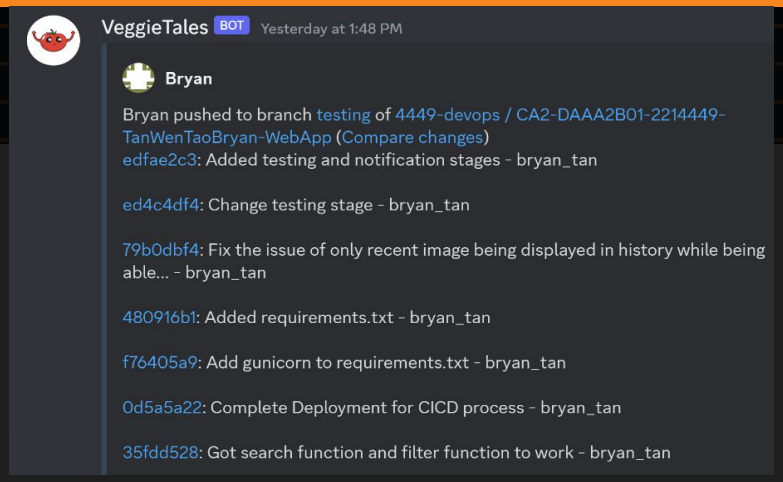
Testing **locally deployed** models

```python
import pytest
import requests
import base64
import json
import numpy as np
from tensorflow.keras.preprocessing import image
import os

# Server URLs (test remote deployment)
url_31x31 = "https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_31x31:predict"
url_128x128 = "https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_128x128:predict"

# Load test images from images folder
def load_image(img_size):
    local_path = os.path.join(os.getcwd(), 'tests/images')
    images_list = []
    for label in os.listdir(local_path):
        for filename in os.listdir(os.path.join(local_path,label)):
            # Load image of specified size and feature scale
            img = image.load_img(os.path.join(local_path,label, filename), color_mode='grayscale', target_size=(img_size, img_size))
            img = image.img_to_array(img)/255.0
            # Reshape image to (1, img_size, img_size, 1)
            img = img.reshape(1, img_size, img_size, 1)
            images_list.append(img)
    return images_list

# Predict using test images
def make_prediction(instances, url):
    # Send POST API request to server
    data = json.dumps({"signature_name": "serving_default", "instances": instances.tolist()})
    headers = {"content-type": "application/json"}
    json_response = requests.post(url, data=data, headers=headers)
    # Parse response
    predictions = json.loads(json_response.text)['predictions']
    return predictions

# Test prediction for 31x31 model
def test_prediction_31():
    data = load_image(31)
    for img in data:
        predictions = make_prediction(img, url_31x31)
        # Check if prediction is a list
        assert isinstance(predictions, list)
        # Check if prediction is a list of length 1
        assert len(predictions) == 1
        # Check if each prediction is a float
        assert isinstance(predictions[0][0], float)

# Test prediction for 128x128 model
def test_prediction_128():
    data = load_image(128)
    for img in data:
        predictions = make_prediction(img, url_128x128)
        # Check if prediction is a list
        assert isinstance(predictions, list)
        # Check if prediction is a list of length 1
        assert len(predictions) == 1
        # Check if prediction is a float
        assert isinstance(predictions[0][0], float)
```

```python
# Server URLs (test local deployment)
url_31x31 = "http://vegetable_cnn:8501/v1/models/veggie_cnn_31x31:predict"
url_128x128 = "http://vegetable_cnn:8501/v1/models/veggie_cnn_128x128:predict"
```



CI/CD Pipeline

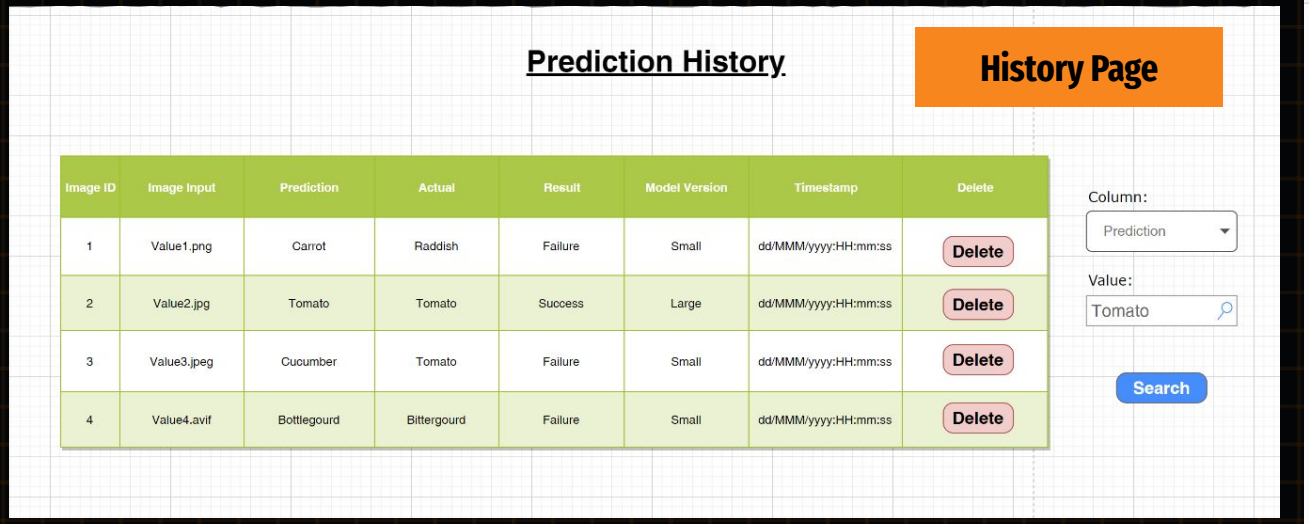


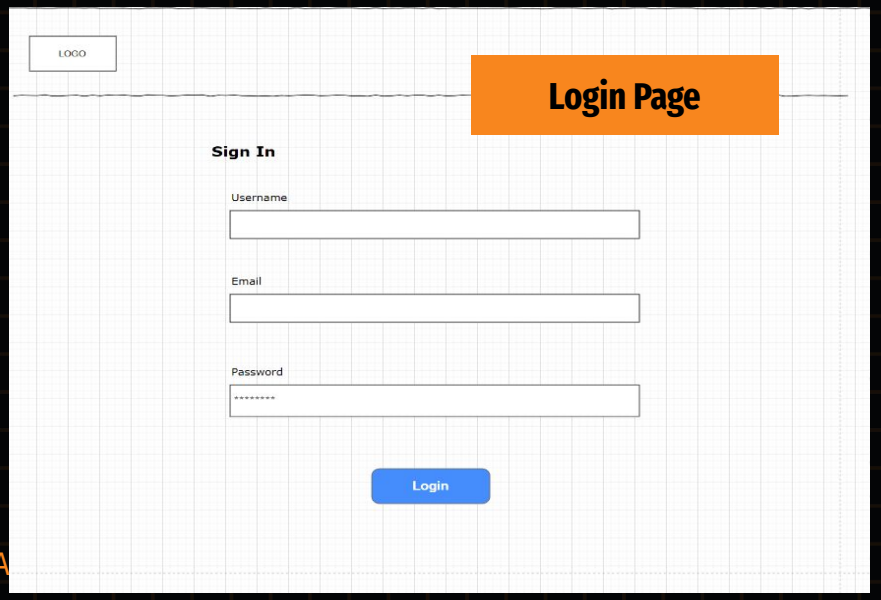

```yaml
stages:
  - test
  - deploy
  - notification

pytest:
  stage: test
  before_script:
    - pwd
  image: python:3.8
  script:
    - pip install -r requirements.txt
    # Skip local deployment test because docker does not store the local files used to run the test in the container
    - python -m pytest -k "not tests/test_docker_local.py" --junitxml=junit.xml
  artifacts:
    reports:
      junit: junit.xml

deployment:
  stage: deploy
  script:
    - curl https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_31x31
    - curl https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_128x128
  only:
    - main

notification:
  stage: notification
  script:
    - 'if [ "$CI_PIPELINE_STATUS" == "success" ]; then curl -X POST -H "Content-type: application/json" --data "{\"content\":\"Pipeline Succeeded: $CI_COMMIT_REF_NAME - $CI_COMMIT_TITLE\"}" https://discord.com/api/webhooks/1200457121326710914/8NBM-ImDrvrtSmNB-Ynay07LBz9H5NvLC64gJSF9W1Vu5DluE-UrulhCuCu28gYy08Hy; fi'
    - 'if [ "$CI_PIPELINE_STATUS" == "failed" ]; then curl -X POST -H "Content-type: application/json" --data "{\"content\":\"Pipeline Failed: $CI_COMMIT_REF_NAME - $CI_COMMIT_TITLE\"}" https://discord.com/api/webhooks/1200457121326710914/8NBM-ImDrvrtSmNB-Ynay07tBz9H5NvLC64gJSF9W1Vu5DluE-UrulhCuCu28gYy08Hy; fi'
```

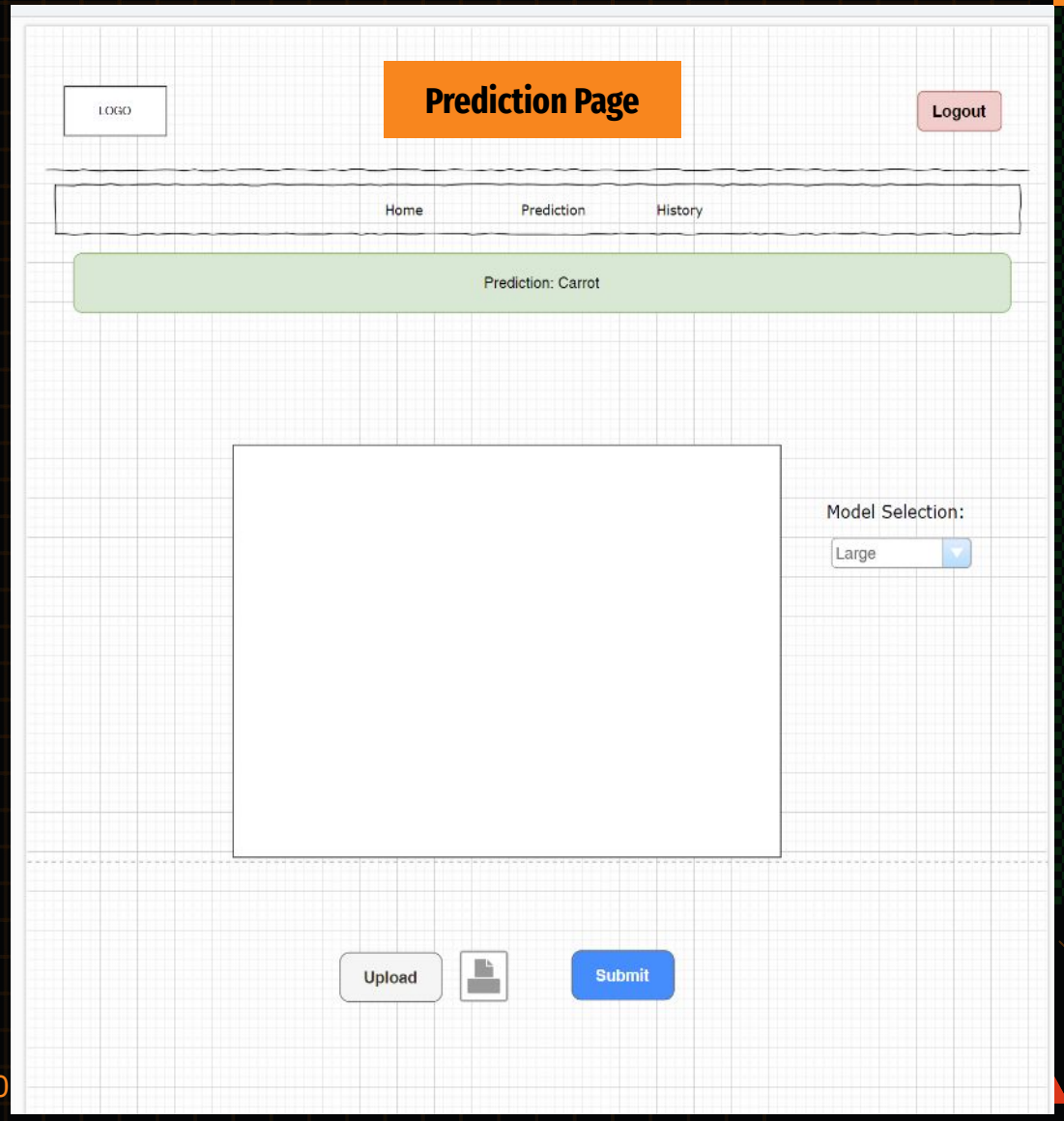
- **3 stages** - Test, Deploy, Notification
- **Continuous Integration** - Each commit and merge triggers an automated series of tests using PyTest to ensure changes do not conflict with existing codebase
- **Continuous Deployment** - Every change that passed automated testing will be automatically deployed to production via Render
- **Feedback Loop** - Developers receive immediate feedback, allowing problems to be addressed quickly

- **Discord -** Channel used to inform developers of new issues, closed issues, commits, merge to production branch through webhooks and integration to GitLab
- Conducted for both **web application** and model development and deployment

Wireframes (Initial Mockup)







forms.py (Prediction & SearchFilter)

Prediction Form

```python
class PredictionForm(FlaskForm):
    file_upload = FileField("Upload Image", validators=[FileRequired(), FileAllowed({'jpg', 'png', 'jpeg'}, message="File Rejected: Only jpg, jpeg, and png files are allowed.")])
    model = SelectField("CNN Model", choices=[("veggie_cnn_31x31", "Veggie CNN (31x31)"), ("veggie_cnn_128x128", "Veggie CNN (128x128)")], default="veggie_cnn_31x31", validators=[InputRequired()])
    submit = SubmitField("Predict")
```

Search and Filter Form

```python
class SearchHistoryForm(FlaskForm):
    search_bar = StringField("Search By")
    filter_by_timestamp = SelectField("Timestamp", choices=[("None", "Sort By:"),("desc", "Latest Prediction"), ("asc", "Earliest Prediction")], default="None")
    filter_by_probability = SelectField("Probability", choices=[("None", "Sort By:"),("desc", "Highest Probability"), ("asc", "Lowest Probability")], default="None")

    # Obtain values for model from the database
    filter_by_model = MultiCheckboxField("Model", choices=[("veggie_cnn_31x31", "Veggie CNN (31x31)"), ("veggie_cnn_128x128", "Veggie CNN (128x128)")])
    filter_by_prediction = MultiCheckboxField("Prediction", choices = [("Bean", "Bean"), ("Bitter Gourd", "Bitter Gourd"), ("Bottle Gourd", "Bottle Gourd"), ("Brinjal", "Brinjal"), ("Broccoli","Broccoli"), ("Cabbage","Cabbage"), ("Capsicum", "Capsicum"),
                                                                        ("Carrot","Carrot"), ("Cauliflower","Cauliflower"), ("Cucumber","Cucumber"), ("Papaya", "Papaya"), ("Potato","Potato"),
                                                                        ("Pumpkin","Pumpkin"), ("Radish","Radish"), ("Tomato", "Tomato")])
```

- **Prediction Form:**
  - Validators include **FileRequired**
  - Allows **only certain file types** (i.e. jpg, png and jpeg) using FileAllowed, returns custom message

- **SearchHistoryForm:** (Search and Filter)
  - Allows users to choose the records returned based on certain filters
  - Accept **multiple filters** like having records with **more than 1 type of predictions** (i.e. Tomato & Carrot) and **models** (i.e. Vegetable CNN 31x31 & Vegetable CNN 128x128)
  - Return records with certain keywords in **search bar**
  - Sort records by **timestamp and probability** in **ascending and descending order**

SQLite Database (models.py)

1-M Relationship:

- **1 user** can have **many entries**
- **Foreign Key** (user_id) in Entry table **reference** to **Primary Key** (id) in User table

```python
class Entry(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    # Stores the user id of user who made the prediction
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

    carsprice_range = db.Column(db.String(14))
    aspiration = db.Column(db.String(5))
    carlength =  db.Column(db.Float)
    curbweight =  db.Column(db.Float)
    horsepower = db.Column(db.Float)
    wheelbase = db.Column(db.Float)
    carbody = db.Column(db.String(11))
    drivewheel = db.Column(db.String(3))
    enginetype = db.Column(db.String(5))
    carwidth = db.Column(db.Float)
    enginesize = db.Column(db.Float)
    boreratio = db.Column(db.Float)
    cylindernumber = db.Column(db.String(5))
    fueleconomy = db.Column(db.Float)

    prediction = db.Column(db.Float)
    predicted_on = db.Column(db.DateTime, nullable=False)
```

For prediction entries

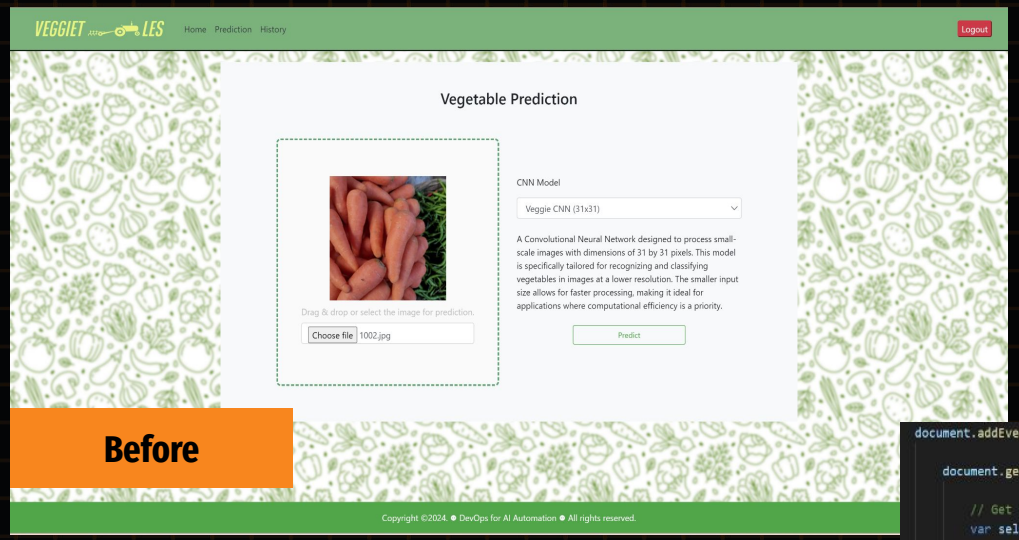
```python
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(20), nullable=False, unique=True)
    email = db.Column(db.String(100), nullable=False, unique=True)
    password = db.Column(db.String(80), nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False)
    # Stores the relationship between user and prediction
    user = db.relationship('Entry', backref='user', lazy=True)
```

For registration & authentication

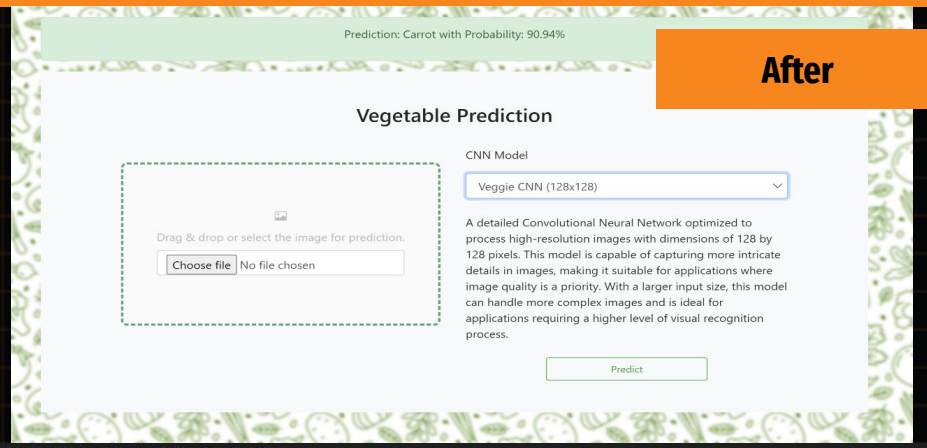
```python
    # Generates a token to reset password
    def get_reset_password_token(self, expires_in=1800):
        encoded = jwt.encode({'reset_password': self.id, 'exp': datetime.utcnow() + timedelta(seconds=expires_in)},
                             app.config['SECRET_KEY'], algorithm='HS256')
        return encoded

    # Function to verify the token and returns the user id
    # Converts function to be static method
    @staticmethod
    def verify_reset_password_token(token):
        try:
            id = jwt.decode(token, app.config['SECRET_KEY'], algorithms=['HS256'])['reset_password']
        except:
            return
        return User.query.get(id)
```

Password Recovery using JWT

Prediction Page



Before



After

- Users can **switch** models, each caters to **different image size**
- **Different models descriptions** shown once model has been selected
- **Prediction** and its **probability** will be flashed above the canvas
- **Images with (i.e. .png, .jpeg, .jpg) extensions** are **accepted** and can be displayed once file is uploaded, **does not accept non-images file**

```
document.addEventListener('DOMContentLoaded', function() {

    document.getElementById('model-select').addEventListener('change', function(){

        // Get the selected model
        var selectedModel = this.value

        var description = document.getElementById('model_description')
        switch(selectedModel) {
            case 'veggie_cnn_31x31':
                description.textContent = "A Convolutional Neural Network designed to process small-scale images with dimensions of 31 by 31 pixels. This model is specifically tailored fo
                break;
            case 'veggie_cnn_128x128':
                description.textContent = "A detailed Convolutional Neural Network optimized to process high-resolution images with dimensions of 128 by 128 pixels. This model is capable
                break;
            default:
                description.textContent = "Select a model to view its description."
        }
    })
})

//Display images when images are uploaded
function displayImages(event){
    const fileInput = event.target;

    const uploadedImages = document.getElementById('preview-image');

    if (fileInput.files && fileInput.files[0]){
        const reader = new FileReader();

        reader.onload = function(e){
            uploadedImages.innerHTML = `<img src="${e.target.result}" alt="Uploaded Image" class="img-fluid" style="max-width: 100%; max-height: 100%;"/>`
        }

        reader.readAsDataURL(fileInput.files[0]);
    }
}
```

Prediction Page

```python
@app.route("/predict", methods=["GET", "POST"])
@cross_origin(origin='localhost',headers=['Content- Type','Authorization'])
def predict():
    form = PredictionForm()
    if request.method == "POST":
        if form.validate_on_submit():
            selected_model = form.model.data
            if selected_model == 'veggie_cnn_31x31':
                size = 31
                url = "https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_31x31:predict"

            else:
                size = 128
                url = "https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_128x128:predict"

            image, image_in_bytes = preprocess_img(form.file_upload.data, size)
            # Reshape the image to the format the model expects to have a single channel
            image = image.reshape((1, size, size, 1))
            # Send POST API request to server
            data = json.dumps({"signature_name": "serving_default", "instances": image.tolist()})
            headers = {"content-type": "application/json"}
            json_response = requests.post(url, data=data, headers=headers)
            # Parse response
            predictions = json.loads(json_response.text)['predictions']
            predictions_class_index = np.argmax(predictions[0])
            # Predicted class
            predicted_class = veggie_classnames[predictions_class_index]
            # Probabilty of prediction
            prob_score = predictions[0][predictions_class_index]*100
            prob_score = round(prob_score, 2)

            # Singapore timezone timestamp
            timezone = pytz.timezone("Asia/Singapore")
            current_time = datetime.datetime.utcnow().replace(tzinfo=pytz.utc).astimezone(timezone)

            new_entry = Entry(image = image_in_bytes, DL_model = selected_model, prediction = predicted_class, probability = prob_score, predicted_on = current_time)
            add_entry(new_entry)

            flash(f"Prediction: {predicted_class} with Probability: {prob_score}%", "success")

        else:
            flash(f"Error: Unable to proceed with prediction", "danger")
    return render_template("index.html", form=form, title="Vegetable Prediction", index=True, entries=get_entries())
```

Predicts the test data based on the models deployed and served in **Render** previously

```python
# Filter entries with the pattern checks to ensure that the entries matched the requirements chosen by the users
@app.route('/filter', methods=['GET'])
def filter_entries(data):
    if data is None:
        data = request.get_json()

    user_id = data["user_id"]
    timestamp_filter = data["timestamp_filter"]
    probability_filter = data["probability_filter"]
    model_filter = data["model_filter"]
    prediction_filter = data["prediction_filter"]
    search_bar = data["search_bar"]
    where_clause = (Entry.user_id == user_id)

    # Check for words within search bar
    if search_bar:
        where_clause = where_clause & (Entry.DL_model.like(f"%{search_bar}%") | Entry.prediction.like(f"%{search_bar}%") | Entry.probability.like(f"%{search_bar}%")) | Entry.predicted_on.like(f"%{search_bar}%")

    # Check for certain models
    if model_filter is not None:
        model_conditions = (Entry.DL_model == model for model in model_filter)
        where_clause = where_clause & (reduce(lambda x,y: x|y, model_conditions))

    # Check for the type of predictions
    if prediction_filter is not None:
        prediction_conditions = (Entry.prediction == prediction for prediction in prediction_filter)
        where_clause = where_clause & (reduce(lambda x,y: x|y, prediction_conditions))

    # Retrieve the entries to update them based on where clause
    entries = get_entries_by_filter(where_clause)

    entries = [
        {
            "id": entry.id,
            "image": base64.b64encode(entry.image).decode('utf-8'),
            "DL_model": entry.DL_model,
            "prediction": entry.prediction,
            "probability": entry.probability,
            "predicted_on": entry.predicted_on.strftime("%d %b %Y %H:%M"),
            "user_id": entry.user_id
        }
        for entry in entries
    ]
```

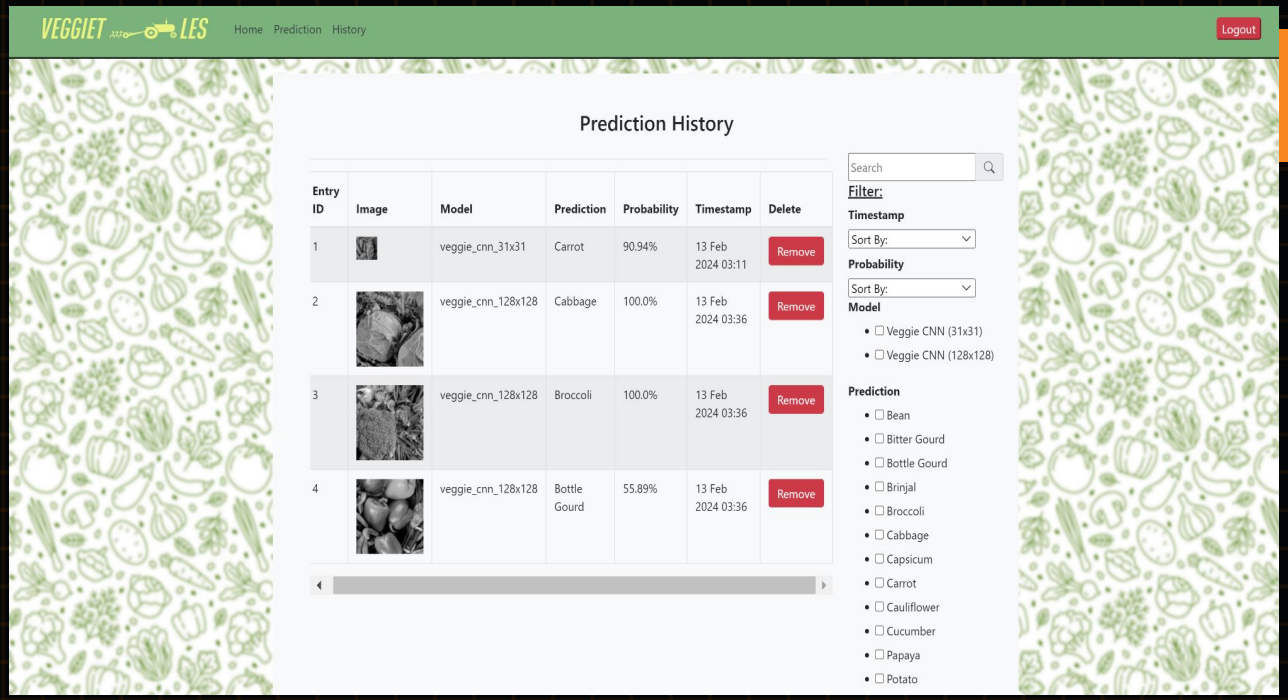
Using **regex** to check for **certain keywords** in **search bar**

Reduce entries such that it matches **filters** for **model and predictions**

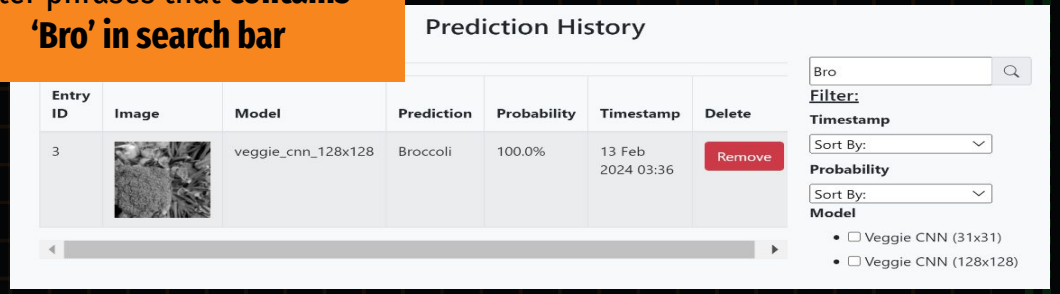
Sort entries according to **timestamp/probability** in **ascending/descending order**

```python
# Check timestamp filter
if timestamp_filter:
    # Return entries according to most recent entries or earliest entries
    if timestamp_filter == "desc":
        entries = sorted(entries, key=lambda x: x["predicted_on"], reverse=True)
    else:
        entries = sorted(entries, key=lambda x: x["predicted_on"])

# Check probability filter
if probability_filter:
    # Return entries according to greatest probability or smallest probability
    if probability_filter == "desc":
        entries = sorted(entries, key=lambda x: x["probability"], reverse=True)
    else:
        entries = sorted(entries, key=lambda x: x["probability"])

if entries is None:
    return jsonify({'error': 'Failed to get entries'})
else:
    # Return the entries
    return jsonify({'entries': entries})
```

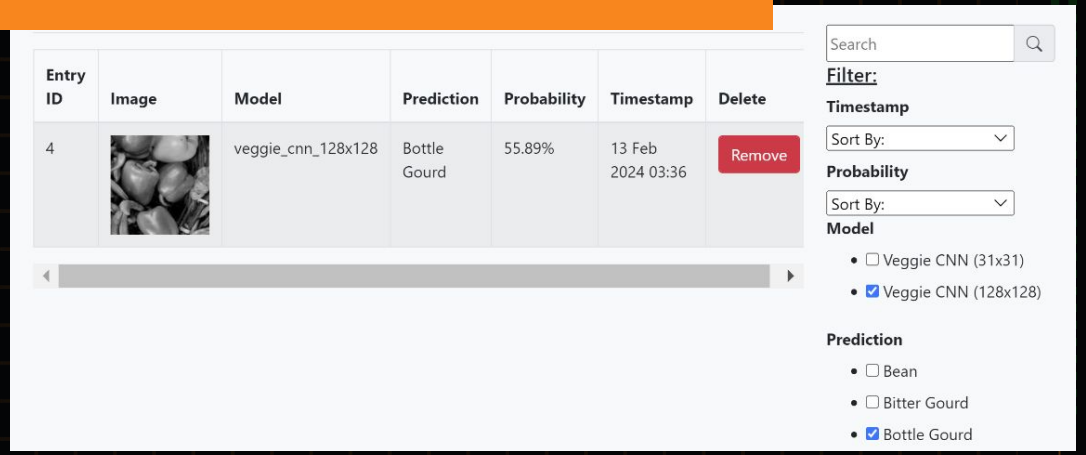
History Page



Filter phrases that **contains 'Bro' in search bar**



Filter records predicted by **Veggie CNN (128x128)** and **Bottle Gourd** in **checkboxes**



- Users can **search and filter records** as listed before
- Images are **displayed** in each record
- Users can **remove entry** when needed

Registration (Flask-Login)

```python
class RegistrationForm(FlaskForm):
    username = StringField("Username", validators=[InputRequired(), Length(min=4, max=20), Regexp(r"^[a-zA-Z0-9_.]*",
                                                                                                  message="Username must only contain letters, numbers, underscores and periods.")])
    email = EmailField("Email", validators=[InputRequired(), Length(min=3, max=100), Regexp(r"[^@]+@[^@]+\.[^@]+",
                                                                                            message='Invalid email address, please use a valid email address that contains an @')])
    password = PasswordField("Password", validators=[InputRequired(), Length(min=8, max=20),Regexp(r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]+",
                                                                                            message='Password must contain at least one uppercase letter, one lowercase letter, 1 special character and 1 numerical character.')])
    confirm_password = PasswordField("Confirm Password", validators=[InputRequired(), EqualTo("password", message="Passwords must match, please try again.")])
    submit = SubmitField("Register")

    # Validates the username
    def validate_username(self, username):
        existing_username = User.query.filter_by(username=username.data).first()
        if existing_username:
            raise ValidationError("That username is taken. Please choose a different one.")

    # Validates the email
    def validate_email(self, email):
        existing_email = User.query.filter_by(email=email.data).first()
        if existing_email:
            raise ValidationError("That email has been registered. Please use a different one or login instead.")
```

```python
@app.route("/register", methods=['GET', 'POST'])
def registration_page():
    form = RegistrationForm()
    # Check if user is already logged in
    if current_user.is_authenticated:
        flash(f"You are already logged in as {current_user.username}", "info")
        return redirect(url_for("home"))

    if form.validate_on_submit():
        # Hash password
        hashed_password = bcrypt.generate_password_hash(form.password.data)

        # Change timezone to Singapore
        timezone = pytz.timezone("Asia/Singapore")
        current_time = datetime.utcnow().replace(tzinfo=pytz.utc).astimezone(timezone)

        # Add user to database
        new_user = User(username=form.username.data, email=form.email.data, password=hashed_password, timestamp=current_time)
        db.session.add(new_user)
        db.session.commit()
        flash(f'Account created for {form.username.data}! Try logging in now.', 'success')
    return render_template("register.html", title="Registration", form=form)
```

Rejects access if user is authenticated

Encode password by hashing it upon submit

Render form by connecting to server

Form Validation (Registration):

- Input Required
- Length (Min & Max No.of Characters)
- Regex (Patterns)
  - Username allowed letters, numbers, underscores and periods
  - Email must have content in front and behind @
  - Password must have at least one uppercase letter, one lowercase letter, 1 special character & 1 number
- Confirmation Password to match Password
- Validation methods to check for existing data

Login (Flask-Login)

checks if **user** exist in **database**

checks if password stored matches **password** in form

```python
# Handles http://127.0.0.1:5000/
@app.route("/")
@app.route("/index")
@app.route("/login", methods=['GET', 'POST'])
def login_page():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()

        if user:
            if bcrypt.check_password_hash(user.password, form.password.data):
                flash(f'Welcome {form.username.data}! You are now logged in,', 'success')
                login_user(user, remember=form.remember.data)
                return redirect(url_for("home"))

        flash('Login Unsucessful. Please check your credentials again.', 'danger')

    return render_template("login.html", title="Login", form=form)

@app.route("/logout")
def logout():
    logout_user()
    return redirect(url_for("login_page"))
```

logout user

requires login to access

```python
@app.route("/prediction")
@login_required
def index_page():
    form1 = PredictionForm()
    return render_template("index.html", form =form1, title="Enter Car Price Parameters")

@app.route("/home")
@login_required
def home():
    return render_template("home.html", title="What are the features used to affect the price of a car?")

@app.route("/history")
@login_required
def history_page():
    form = PredictionForm()
    return render_template("history.html", title="Prediction History", form=form, entries=get_entries())
```

```python
class LoginForm(FlaskForm):
    username = StringField("Username", validators=[InputRequired(), Length(min=4, max=20)])
    password = PasswordField("Password", validators=[InputRequired(), Length(min=8, max=20)])
    remember = BooleanField("Remember Me")
    submit = SubmitField("Login")
```

allows local storage to **remember** user

```
<!-- Inherits from layout.html -->
{% extends "layout.html" %}
<!-- The block content replace the one encapsulated in layout.html -->
{% block content %}
<script>
    document.addEventListener('DOMContentLoaded', function(){
        const passwordInput = document.querySelector('.password-input')
        const togglePassword = document.querySelector('.toggle-password')

        if (passwordInput && togglePassword){
            togglePassword.addEventListener('click', function(){
                const type = passwordInput.getAttribute('type') === 'password' ? 'text':'password';
                passwordInput.setAttribute('type', type);
            })
        }
    })
</script>
```

Toggle password between **text and hashed password** for **ease of readability**

```python
from flask import Flask, current_app
from flask_mail import Mail
import pickle
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt
from flask_login import LoginManager

#instantiate SQL Alchemy to handle db process
db =SQLAlchemy()

#create the Flask app
app = Flask(__name__)

#instantiate Bcrypt to handle password hashing
bcrypt = Bcrypt(app)

#instantiates login manager to handle user login
login_manager = LoginManager(app)
login_manager.init_app(app)
login_manager.login_view = 'login_page'
```

```python
# Define user loader callback for Flask-login
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

Instantiating Bcrypt and LoginManager

Password-Recovery (Flask-Mail)

```python
# Initiate mail to handle email sending
app.config["MAIL_SERVER"] = "smtp.gmail.com"
app.config["MAIL_PORT"] = 465
app.config["MAIL_USERNAME"] = "daaa2b01.2214449.tanwentaobryan@gmail.com"
app.config["MAIL_PASSWORD"] = "sqhrubshwgeueige"
app.config["MAIL_USE_TLS"] = False
app.config["MAIL_USE_SSL"] = True
```

```python
@app.route("/forgot_password", methods=['GET', 'POST'])
def forgot_password_page():
    form = PasswordResetRequestForm()
    if form.validate_on_submit():
        # Checks if email exist in the database
        user = User.query.filter_by(email = form.email.data).first()
        if user:
            send_password_reset_email(user)
            flash("An email has been sent with instructions to reset your password.", "info")
            return redirect(url_for("login_page"))
        flash("Unregistered email. Please try again.", 'danger')

    return render_template("forgot_password.html", title="Forgot Password", form=form)

@app.route("/reset_password/<token>", methods=["GET", "POST"])
def reset_password(token):
    user = User.verify_reset_password_token(token)
    if not user:
        flash("Invalid or expired token. Please try again.", "danger")
        return redirect(url_for("foreget_password_page"))
    form = PasswordResetForm()
    if form.validate_on_submit():
        # Hash password
        hashed_password = bcrypt.generate_password_hash(form.password.data)
        user.password = hashed_password
        db.session.commit()
        flash("Your password has been updated. You are now able to login.", "success")
        return redirect(url_for("login_page"))
    return render_template("reset_password.html", title="Reset Password", form=form)
```

Uses SMTP to secure a port 465 between **Flask and Gmail**. Enable 2FA to **generate a password** such that Flask can access my gmail to send the link to the **user's email** with a link they can use for **password recovery** for **secure password update**.

Password Request Form:

- Verifies user by **validating** user's **email** keyed in matches the one stored in the **database**

Password Reset Form:

- Creates more **secure** password resets with **JWT** tokens, user can't forge tokens
- **Verifies token** by checking if it matches
- **Hash password** that user key in and commit to database

Password-Recovery (Flask-Mail)

```python
# Generates a token to reset password
def get_reset_password_token(self, expires_in=1800):
    encoded = jwt.encode({'reset_password': self.id, 'exp': datetime.utcnow() + timedelta(seconds=expires_in)},
                         app.config['SECRET_KEY'], algorithm='HS256')
    return encoded

# Function to verify the token and returns the user id
# Converts function to be static method
@staticmethod
def verify_reset_password_token(token):
    try:
        id = jwt.decode(token, app.config['SECRET_KEY'], algorithms=['HS256'])['reset_password']
    except:
        return
    return User.query.get(id)
```

```python
# Send password reset email
def send_password_reset_email(user):
    token = user.get_reset_password_token()
    msg = Message("Password Reset Request", recipients=[user.email], sender="daaa2b01.2214449.tanwentaobryan@gmail.com")
    msg.body = f'''
    Hi, {user.username}!

    To reset your password, visit the following link:
    {url_for("reset_password", token=token, _external=True)}

    CarGuru Team thanks you for using our service!

    If you did not make this request then simply ignore this email and no changes will be made.
    '''
    mail.send(msg)
```

Top:
1) Encode the token.
Stored on the client-side, and it is used to authenticate all requests until it expires.
Once the token expires, the user has to generate another one, and the cycle continues.
2) Decode the token to verify token and return the userid

Top:
1) Sends message to recipient from user with the token for the user to use to update his/her password

Bottom:
1) Same sort of validation for email and password

```python
class PasswordResetRequestForm(FlaskForm):
    email = EmailField("Email", validators=[InputRequired(), Length(min=3, max=100), Regexp(r"[^@]+@[^@]+\.[^@]+",
                                                                                            message='Invalid email address, please use a valid email address that contains an @')])
    submitField = SubmitField("Request Password Reset")

class PasswordResetForm(FlaskForm):
    password = PasswordField("New Password", validators=[InputRequired(), Length(min=8, max=20),Regexp(r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]+",
                                                                                                        message='Password must contain at least one uppercase letter, one lowercase letter, 1 special character and 1 numerical character.')])
    confirm_password = PasswordField("Confirm New Password", validators=[InputRequired(), EqualTo("password", message="Passwords must match, please try again.")])
    submit = SubmitField("Reset Password")
```

Password Request Form & Password Reset Form

Automated Testing

```
========== 95 passed, 13 xfailed, 4 xpassed in 23.77s ========

====================== test session starts ==================
```

```python
# Test filter entries API based on specific emails
@pytest.mark.xfail(reason="Invalid user id")
@pytest.mark.parametrize("filterEntriesList",
    [   # Inserting test entries to test the filters
        [[BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Carrot", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Cauliflower", 58.62]
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Tomato", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Cauliflower", 58.62],
        ["Radish"], "veggie_cnn_31x31"],
        [[BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Carrot", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Radish", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Tomato", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Cauliflower", 58.62],
        ["Radish"], "veggie_cnn_128x128"],
        [[BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Carrot", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Radish", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Tomato", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Cauliflower", 58.62],
        ["Radish", "Cauliflower", "Tomato"], "veggie_cnn_31x31"],
        [[BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Carrot", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Radish", 93.22],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Broccoli", 58.62],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Papaya", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Tomato", 58.62],
        [BytesIO(Image.fromarray(np.zeros((128, 128))).tobytes()).getvalue(), "veggie_cnn_128x128", "Tomato", 93.22],
        [BytesIO(Image.fromarray(np.zeros((31, 31))).tobytes()).getvalue(), "veggie_cnn_31x31", "Cauliflower", 58.62],
        ["Radish", "Cauliflower", "Tomato"], ["veggie_cnn_31x31", "veggie_cnn_128x128"]]
    ])
```

- Create **RESTFUL APIs** for testing to demonstrate the features and APIs are working with the use of PyTest
- **Web Application**: login, registration, predict, add_entry, remove_entry, get_entries, email_verification, filter_entries
- **Model Deployment**: ensure prediction works for local deployment and remote deployment

Test Filter Entries
(Case Scenario)

Return records that **filters certain predictions and models**

Test Filter Entries
(Filter Predictions & Models)

```python
def test_filter_predictions(client, capsys, filterEntriesList, test_client):
    # Use a random user id that was used to testing
    user_id = test_client['userid']

    # Singapore timezone timestamp
    timezone = pytz.timezone("Asia/Singapore")
    current_time = datetime.datetime.utcnow().replace(tzinfo=pytz.utc).astimezone(timezone)
    with capsys.disabled():
        for entry in filterEntriesList:
            if type(entry) != list or len(entry) == 0 or type(entry[-1]) != float:
                continue

            data = {
                "image": entry[0].decode('utf-8'),
                "DL_model": entry[1],
                "prediction": entry[2],
                "probability": entry[3],
                "predicted_on": current_time.strftime("%d %b %Y %H:%M"),
                "user_id": user_id
            }

            res = client.post('/api/post_entries', data=json.dumps(data), content_type='application/json')

            assert res.status_code == 200
            assert res.headers["Content-Type"] == "application/json"

        # Get the predictions later on
        url2 = "/api/filter_entries"
        data2 = {
            "user_id" : user_id,
            "timestamp_filter" : None,
            "probability_filter" : None,
            "model_filter" : filterEntriesList[-1],
            "prediction_filter" : filterEntriesList[-2],
            "search_bar": ""
        }

        res2 = client.get(url2, data=json.dumps(data2), content_type='application/json')

        assert res2.status_code == 200
        for pred in res2.json['entries']:
            assert pred["DL_model"] == filterEntriesList[-1] or pred["prediction"] in filterEntriesList[-2]
```

Consistency Testing

```python
# Consistency Test + GET API Test for Predictions
@pytest.mark.parametrize("predConsistencyList",
    [
        [[np.zeros((1, 31, 31, 1)), "serving_default", "veggie_cnn_31x31"],
         [np.zeros((1, 31, 31, 1)), "serving_default", "veggie_cnn_31x31"],
         [np.zeros((1, 31, 31, 1)), "serving_default", "veggie_cnn_31x31"]],
        [[np.zeros((1, 128, 128, 1)), "serving_default", "veggie_cnn_128x128"],
         [np.zeros((1, 128, 128, 1)), "serving_default", "veggie_cnn_128x128"],
         [np.zeros((1, 128, 128, 1)), "serving_default", "veggie_cnn_128x128"]]
    ]
)
def test_predict_GET_API(client, predConsistencyList, capsys):
    with capsys.disabled():
        predictedOutput = []
        probabilityOutput = []
        for predictions in predConsistencyList:
            data = {
                'image': predictions[0].tolist(),
                'signature_name': predictions[1],
                'model': predictions[2]
            }
            response = client.get('/api/predict', data=json.dumps(data), content_type='application/json')
            response_body = response.json

            # Check if the response is valid
            assert response.status_code == 200
            assert response_body['prediction']
            predictedOutput.append(response_body['prediction'])

            assert response_body['probability']
            probabilityOutput.append(response_body['probability'])

            # Check if the prediction is consistent
            assert len(set(predictedOutput)) == 1
            assert len(set(probabilityOutput)) == 1

            # Make sure all the predictions are the same
            assert all(x == predictedOutput[0] for x in predictedOutput)
            assert all(x == probabilityOutput[0] for x in probabilityOutput)
```

Predict API + GET API

```python
# Predict API - test prediction
@app.route('/api/predict', methods=["GET", "POST"])
def predict_api():
    data = request.get_json()

    # Retrieve each field from data
    selected_model = data["model"]
    instances = data["image"]

    signature_name = data["signature_name"]

    if selected_model == 'veggie_cnn_31x31':
        url = "https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_31x31:predict"
    else:
        url = "https://veggietales-cnn.onrender.com/v1/models/veggie_cnn_128x128:predict"

    # Send POST API request to server
    data = json.dumps({"signature_name": signature_name, "instances": instances})
    headers = {"content-type": "application/json"}
    json_response = requests.post(url, data=data, headers=headers)
    # Parse response
    predictions = json.loads(json_response.text)['predictions']
    predictions_class_index = np.argmax(predictions[0])

    # Predicted class
    predicted_class = veggie_classnames[predictions_class_index]

    # Probabilty of prediction
    prob_score = predictions[0][predictions_class_index]*100
    prob_score = round(prob_score, 2)

    # Return the prediction in response
    return jsonify({'status': 'success', 'prediction': predicted_class, 'probability': prob_score, 'status_code': 200})
```

Test if **predictions** and **probabilities** of both model are consistent

- Check **consistency** of application behaviour
- Data is grouped into **3 arrays** and looped into the **ML model** to predict and check if all **predictions** into the **3 arrays are equal**
- Make use of /api/predict API

Expected Failure Testing

```python
# Function for registering user to database
def register_user_API(user, email, password):
    try:
        # Hash password
        hashed_password = bcrypt.generate_password_hash(password)

        # Change timezone to Singapore
        timezone = pytz.timezone("Asia/Singapore")
        current_time = datetime.datetime.utcnow().replace(tzinfo=pytz.utc).astimezone(timezone)

        # Add user to database
        new_user = User(username=user, email=email, password=hashed_password, timestamp=current_time)
        db.session.add(new_user)
        db.session.commit()

        # Return the id of the user added
        return new_user.id

    # Registration failure
    except Exception as e:
        db.session.rollback()
        flash(e, "danger")
        return None

# Register API - test user registration
@app.route('/api/register', methods=["GET","POST"])
def register_api():
    data = request.get_json()

    # Retrieve each field from data
    username = data['username']
    email = data['email']
    password = data['password']
    confirm_password = data['confirm_password']

    # Check if password do not match the confirmed password
    if password != confirm_password:
        return jsonify({'status': 'error','registered': False ,'message': 'Passwords do not match.'})

    # Register the user
    new_user = register_user_API(username, email, password)

    if new_user is not None:
        # Return the userid in response
        return jsonify({'status': 'success', "registered": True, 'userid': new_user})
    else:
        return jsonify({'status': 'error', "registered": False})
```

```python
# Expected Failure Test + POST API Test for Register
@pytest.mark.xfail(reason="Not valid username, email, password or mismatched confirm password")
@pytest.mark.parametrize("registerList",[
    ["testuser@1", "testuser1@gmail.com", "Testuser1!", "Testuser1!", False],
    ["Testuser 2", "testuser2@gmail.com", "Testuser2!", "TestUser2!", False],
    ["testuser.3", "@gmail.com", "Testuser3!", "Testuser3!", False],
    ["testuser4", "testuser4@", "Testuser4!", "Testuser4!", False]
])
def test_register_Failure_POST_API(client, registerList, capsys):
    test_register_Valid_POST_API(client, registerList, capsys)
```

- Verifies if application handles **error or failures** appropriately
- Same function used as the **expected failure testing,** register_user_API()
- **Expects failures** as the result

Validity Testing

```python
# Function for retrieving entry from database based on user_id
def get_entries_API(user_id):
    try:
        # Retrieve all entries from database
        entries = db.session.execute(db.select(Entry).filter(Entry.user_id==user_id).order_by(Entry.id)).scalars()
        return entries
    except Exception as error:
        db.session.rollback()
        flash(error,"danger")
        return []

# Retrieve entry API test retrieving entry from database based on user_id
@app.route('/api/get_entries', methods=['GET'])
def get_api():
    # Retrieve data from request
    data = request.get_json()

    # Retrieve each field from data
    user_email = data['user_email']

    # Retrieve the user id from database
    user_id = User.query.filter_by(email = user_email).first().id

    # Retrieve all entries from database
    new_result = get_entries_API(user_id)
    new_results = []
    # Serialise the entries retrieved in JSON
    for entry in new_result:
        # Stores the user's email in the variable
        email = User.query.filter_by(id = entry.user_id).first().email
        new_results.append({'entry_id': entry.id,'user_email': email ,'prediction': entry.prediction, 'probability': entry.probability})

    # Return the entries retrieved
    return jsonify({'entries': new_results})
```

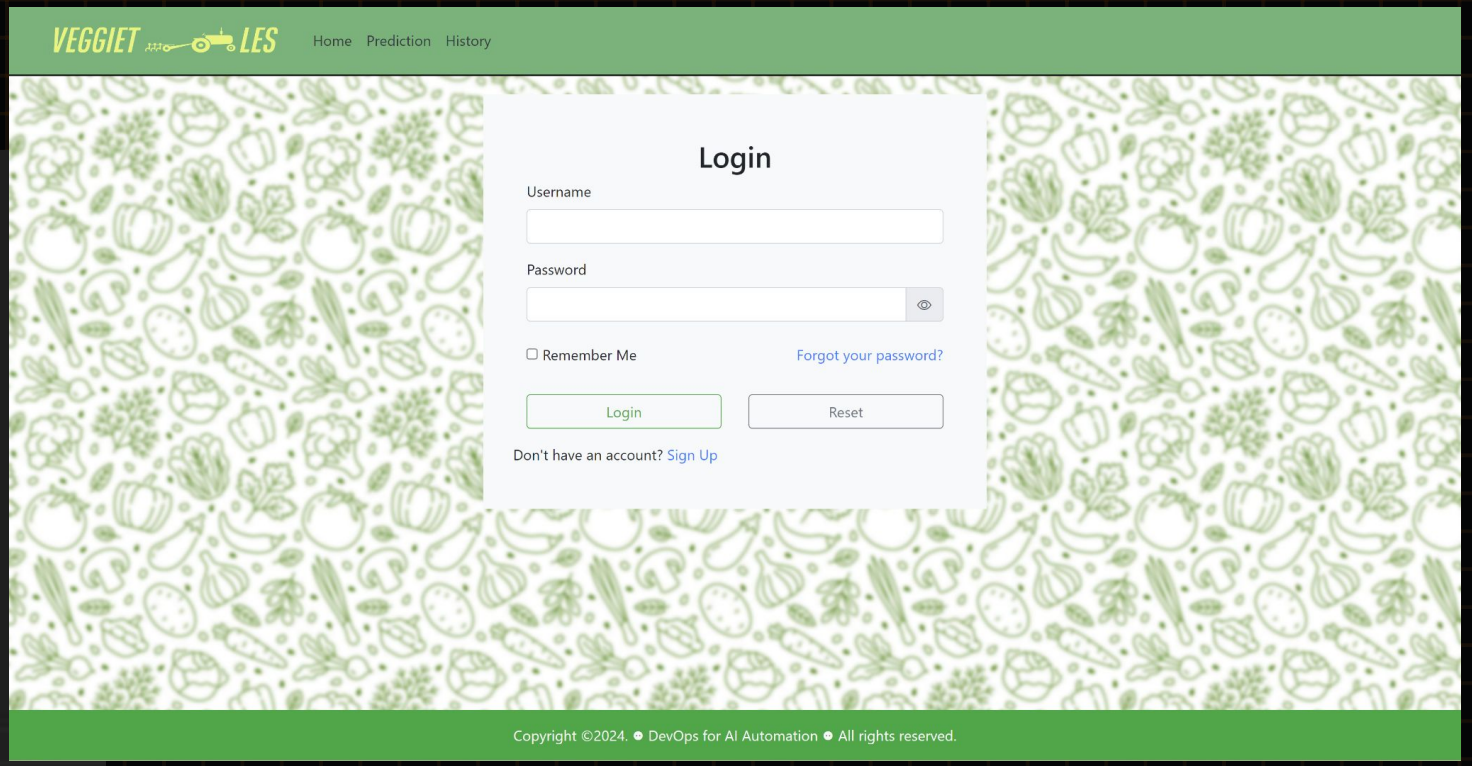
```python
# Test Get Entries API based on specific emails tested previously
@pytest.mark.parametrize("getEntriesList",
    [
        ["Radish", 93.22, "testuser1@gmail.com"],
        ["Carrot", 58.62, "testuser2@gmail.com"],
        ["Radish", 93.22, "testuser3@gmail.com"]
    ]
)
def test_getEntries_API(client, getEntriesList, capsys):
    with capsys.disabled():
        data1 = {
            'user_email': getEntriesList[2]
        }
        response = client.get('/api/get_entries', data=json.dumps(data1), content_type = 'application/json')
        # Check if the response is valid
        assert response.status_code == 200
        assert response.headers["Content-Type"] == 'application/json'
        response_body = json.loads(response.get_data(as_text=True))
        for entry in response_body["entries"]:
            # Check if the response retrieved from the correct user and correct entries are being obtained
            assert entry["prediction"] == getEntriesList[0]
            assert entry["probability"] == getEntriesList[1]
            assert entry["user_email"] == getEntriesList[2]
```

- Ensures that input data is **valid** and meet the **expected criteria** at the end
- **Validate** the data beforehand
- Make use of /api/get_entries to ensure that the entries previously **added** by users of the **particular email**, matches the results to show that the records are being **fetched successfully**

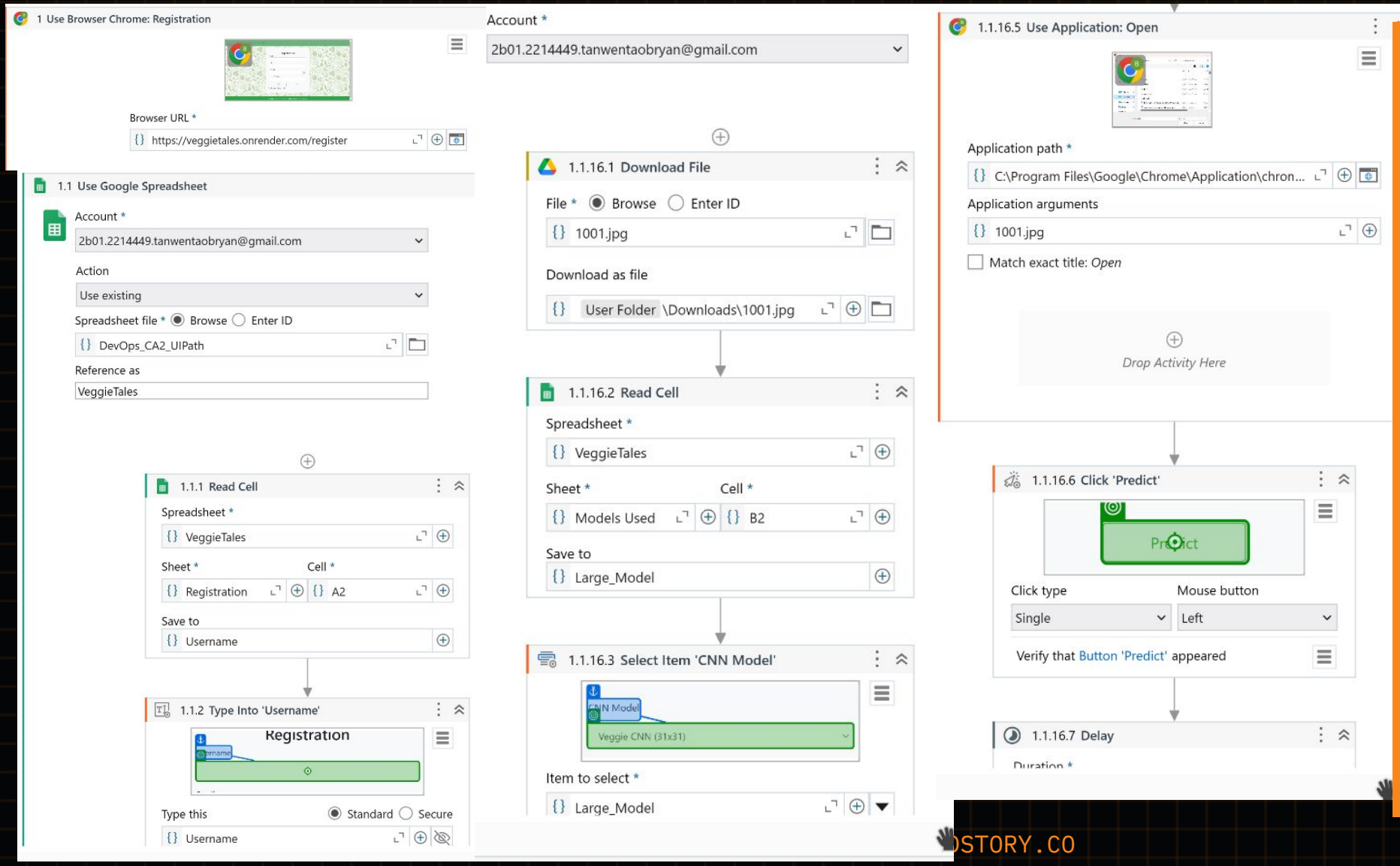
Web App Deployment

- Deploy web app on render with the containerized application and dependencies
- https://veggietales.onrender.com/

```
FROM python:3.8-slim
#update the packages installed in the image
RUN apt-get update -y
# Make a app directory to contain our application
RUN mkdir /app
# Copy every files and folder into the app folder
COPY . /app
# Change our working directory to app fold
WORKDIR /app
# Install all the packages needed to run our web app
RUN pip install -r requirements.txt
# Add every files and folder into the app folder
ADD . /app
# Expose port 5000 for http communication
EXPOSE 5000
# Run gunicorn web server and binds it to the port
CMD gunicorn --bind 0.0.0.0:5000 app:app
```

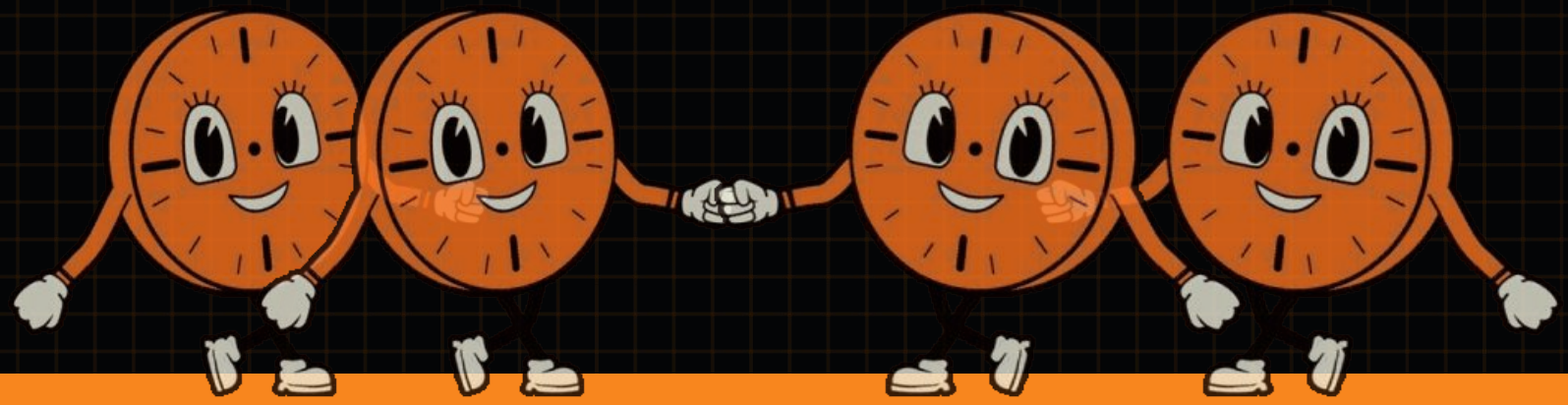


Robotic Process Automatic



Use UIPath to **automate GUI testing**

- **Registration and Login**
  - **'Google Spreadsheet'** used to **store values used** to fill in the form
  - Store values in **variables** to use in the next activity
  - Read as cell and use **'Type Into' and 'Click'** activities to ensure details are processed
- **Predict**
  - Using **GDrive** to **store images** and **'Download File'** and select the image to be **uploaded into prediction**



~THANK YOU~