ParkerAI

DOCUMENTATION

Shai Shillo & Guy David

Table of Contents

Abstract

The CPPS project aims to improve urban mobility by predicting parking space availability, thus easing parking experiences and decreasing traffic and environmental impact. It involves a server application that processes real-time data from parking lots using machine learning and computer vision (PyTorch, OpenCV) to predict occupancy and train models, API to publish and control the data flow with client authentication, python-telegram-gui client that consumes the data and show it to the end user. On server and API Node.js is used to create the server that coordinates the system's components and communicates with API and Clients. On client Python is used to create the views.

The Project features modules for image processing, machine learning data preparation, a socket server for communication, debugging tools, and a logging system for reliability Its innovative integration of machine learning and web technologies offers a smart parking solution with the potential for accurate predictions, benefiting city planners, operators, and drivers. The project is designed for adaptability and scalability.

Intro Modules

The project is built from three main modules:

CPPS-Server

manages the process of data collection it is an Express.js server that capture image using Termux:API , rotate (using ImageMagic) then image is crop (using Python-OpenCV) to a smaller image based on a Blueprint (Process), Predict (using PyTorch MobileNetV3 Large) and Store the data to MongoDB.
The CPPS-Server have 3 modes: start, debug, learn.

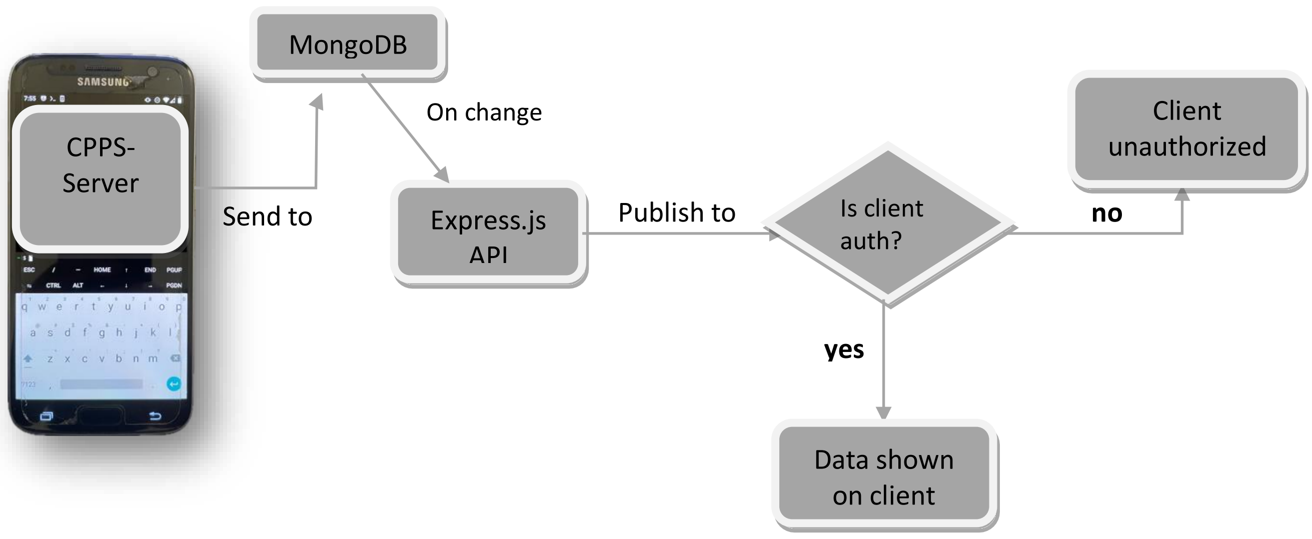
API

is an Express.js API that controls the data after each CPPS cycle, subscribed clients (using a secured protocol) can consume the published data through this module.

Telegram-bot-gui

a python basic GUI that based on python-telegram-bot package. The client subscribes to the API, when API publish a new data it consumes it, build an image to illustrate parking slot and then show it to the end user based on user requests.

Figure 1: Basic workflow when CPPS-Server is on Start mode.



Project Overview

Technologies Used

Server

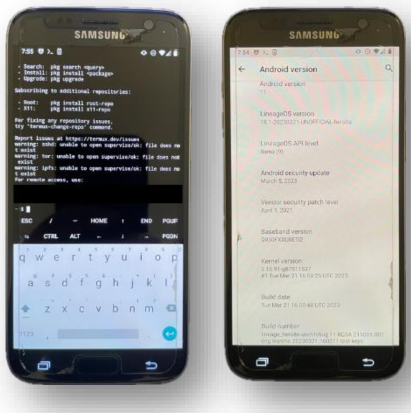
Smart Phone

Smart Phones are a great SoC and more than that, they include cameras, sensors and connectivity components which valuable for the project. In our case a galaxy s7.

Android OS

Android OS is an Open Source Operation System which is suitable for the project needs hence we want to expose Android functionality. in our case android 11 (LingeOS 18.1).

Termux



```
7:55
- Search:  pkg search <query>
- Install: pkg install <package>
- Upgrade: pkg upgrade

Subscribing to additional repositories:

- Root:    pkg install root-repo
- X11:     pkg install x11-repo

For fixing any repository issues,
try 'termux-change-repo' command.

Report issues at https://termux.dev/issues
warning: sshd: unable to open supervise/ok: file does no
t exist
warning: tor: unable to open supervise/ok: file does not
 exist
warning: ipfs: unable to open supervise/ok: file does no
t exist
For remote access, use:
[redacted]
~ $
```

Android version

Android version
11

LineageOS version
18.1-20230321-UNOFFICIAL-herolte

LineageOS API level
Ilama (9)

Android security update
March 5, 2023

Vendor security patch level
April 1, 2021

Baseband version
G930FXXU8ETI2

Kernel version
3.18.91-g87511537
#1 Tue Mar 21 16:08:25 UTC 2023

Build date
Tue Mar 21 16:00:48 UTC 2023

Build number
lineage_herolte-userdebug 11 RQ3A.211001.001 eng.leanne.20230321.160211 test-keys

Figure 2

Termux is a free and open-source terminal emulator.

for Android which allows for running

a Linux environment on an Android device.

Termux include 7 add-ons one of them is

Termux: API which exposes Android functionality to CLI applications.

Language

JS and Python are the main code languages, there is small parts of shell script hence we use Termux: API.

Framework

Node.js and Express.js

Machine Learning

To ascertain parking occupancy, we trained a neural network based on the MobileNetV3 Large architecture within the PyTorch framework, utilizing a dataset that we have compiled.
Autonomous process collects and classify (using pre-trained YOLOv4) Data for MobileNetV3 Large
training.

Database

MongoDB database offering flexibility and scalability. the Design Pattern is One-To-Many.
Google Drive for the end user simplicity and learn mode.

Design Pattern:

The architecture and design patterns of the CPPS Server reflect a modular, and event-driven approach, integrating web technologies with machine learning for real-time data processing. This design facilitates scalability, maintainability, and efficient data handling.

API

Language: JavaScript
Framework: Node.js and Express.js For building a fast and scalable server-side application.
database: MongoDB database offering flexibility and scalability.
Connectivity: WebSocket for real-time communication between the API and clients.
Logging: Winston used for logging API activities.
Security: JWT used for generating Tokens so that only authorized clients can subscribe to API

Architecture:

- ***RESTful API:*** The API follows REST principles for creating scalable web services.
- ***Microservices Architecture:*** Modular approach for independent service scalability and maintenance.
- ***Event-Driven Architecture:*** Utilizing WebSockets for real-time data streaming.

Client

Language: Python, XML.
Packages:

- Telegram Bot API for creating and managing the bot interface
- Pillow for build illustration of the parking slots

Connectivity: WebSocket for real-time communication between the API and telegram-bot-gui.

Architecture:

Event-Driven Architecture*:* Utilizes WebSockets for real-time updates.
Modular Design*:* Separate modules for bot interaction (`telegram-bot.py`), UI generation (`ui.py`), and image generation (`generate_image.py`).

Flowcharts

CPPS-Server main entry points

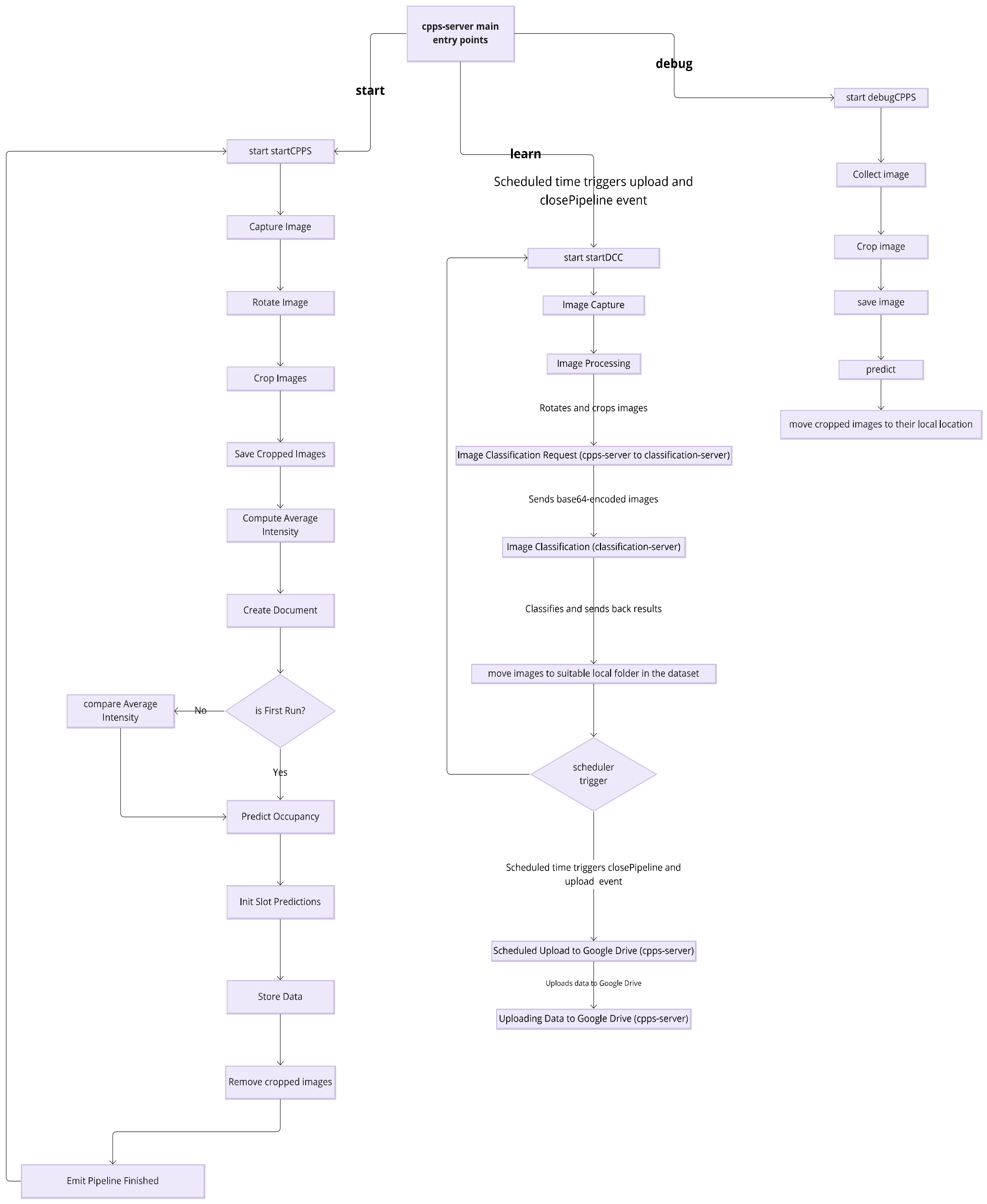

cpps-server main entry points
start
learn
debug
start startCPPS
Capture Image
Rotate Image
Crop Images
Save Cropped Images
Compute Average Intensity
Create Document
is First Run?
No
compare Average Intensity
Yes
Predict Occupancy
Init Slot Predictions
Store Data
Remove cropped images
Emit Pipeline Finished
Scheduled time triggers upload and closePipeline event
start startDCC
Image Capture
Image Processing
Rotates and crops images
Image Classification Request (cpps-server to classification-server)
Sends base64-encoded images
Image Classification (classification-server)
Classifies and sends back results
move images to suitable local folder in the dataset
scheduler trigger
Scheduled time triggers closePipeline and upload event
Scheduled Upload to Google Drive (cpps-server)
Uploads data to Google Drive
Uploading Data to Google Drive (cpps-server)
start debugCPPS
Collect image
Crop image
save image
predict
move cropped images to their local location

Figure 3: cpps-server main entry points

API

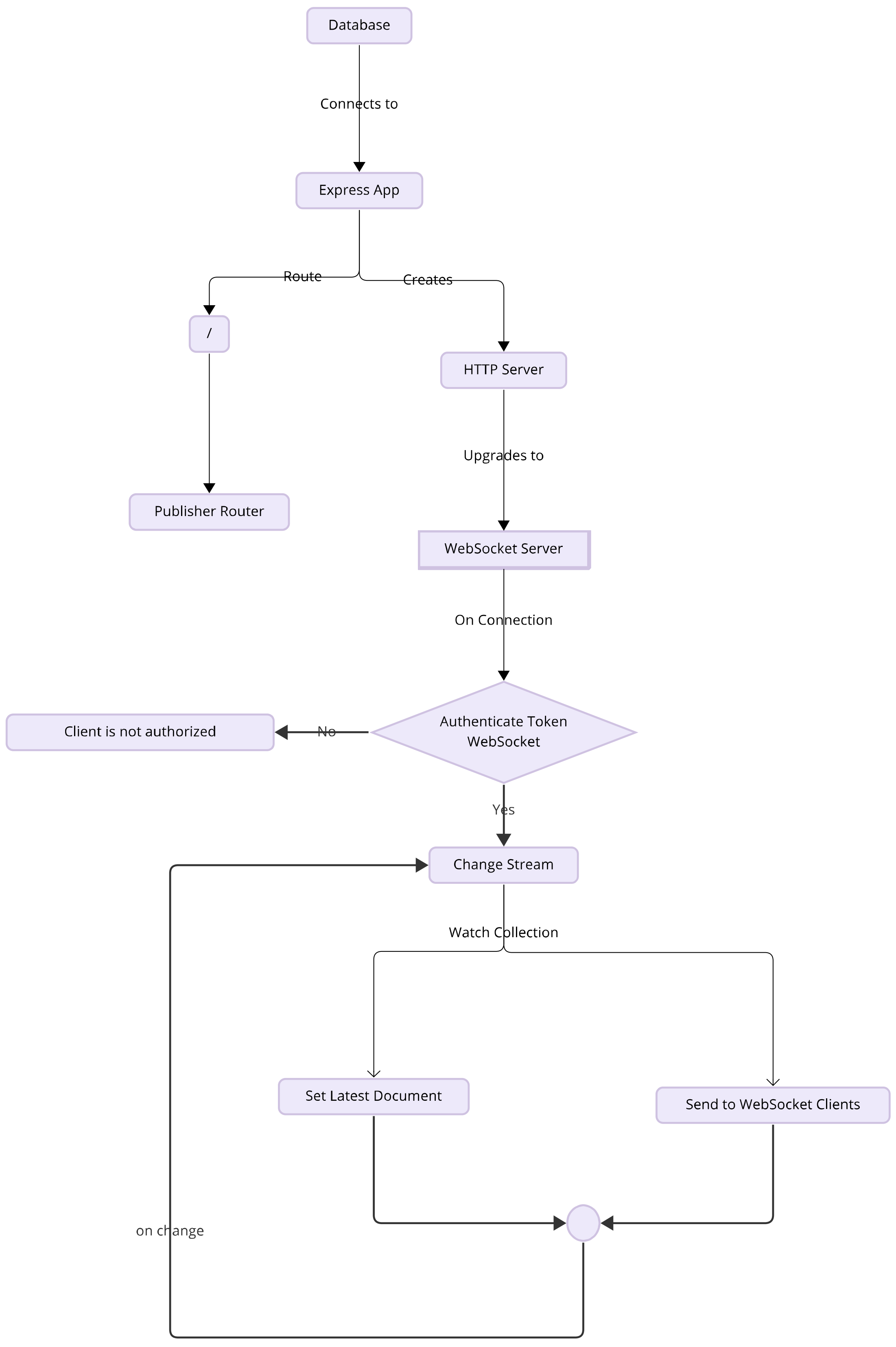


Figure 4: API

Telegram-bot-gui

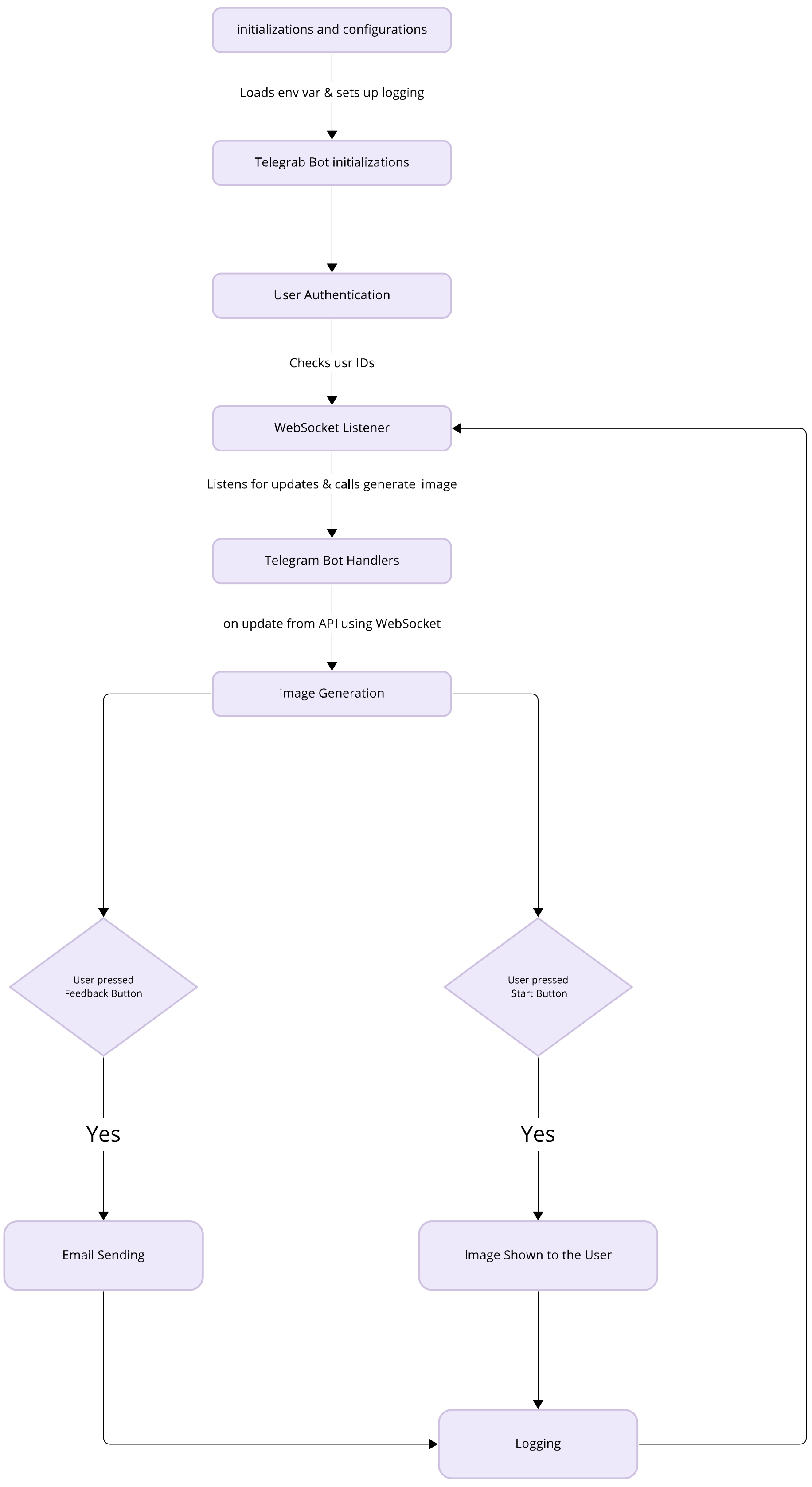

initializations and configurations
Loads env var & sets up logging
Telegrab Bot initializations
User Authentication
Checks usr IDs
WebSocket Listener
Listens for updates & calls generate_image
Telegram Bot Handlers
on update from API using WebSocket
image Generation
User pressed Feedback Button
User pressed Start Button
Yes
Yes
Email Sending
Image Shown to the User
Logging

Figure 5: Telegram-bot-gui

Machine Learning

We started to research the field of ML using our knowledge from ML Academic course, our closest friend Google advised us on pre-trained YoloVX models here is a first look:

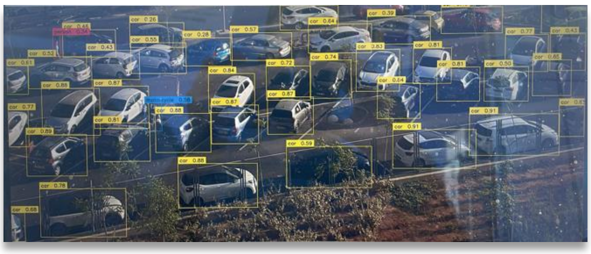


Figure 6: YOLOV8x Prediction Sample

While it successfully predicted the presence of vehicles, it fell short in addressing our primary inquiries – specifically, *the criteria distinguishing between occupied and unoccupied parking slots*. Furthermore, we need insights into the most suitable architecture for real-time prediction on mobile devices

What defines Occupied or Unoccupied parking lot? As we went down the road we decided to go with methodology of if there is a vehicle in a specific parking lot or the absence of one.

Which Architecture is Suitable for Predicting on Mobile device in real-time? The MobileNetV3 Large is a light Architecture that suitable for mobile device due to its balance between accuracy and performance (low latency).

I. **Data Collection:**
   - **Data Capture:** JPG file using mobile phone camera

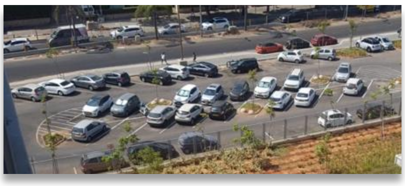


Figure 7: raw data

II. **Data Preparation:**
   - **Crop:** using boundary boxes we crop the image to parking slots
   - **Labeling**: at first classify the cropped images using YOLOV4 pre-trained model then human being will do the fine tune.

III. Note: stage 1 & 2 are autonomous and used as an outside service that we Developed. The boundary boxes are created using CVAT tool.

   - **Dataset:** the dataset classes are occupied and unoccupied, it contains 15,067 occupied and 936 unoccupied images and arranged in folders

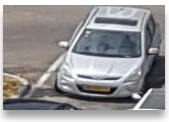


Figure 8: occupied labeled image.

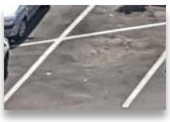


Figure 10: unoccupied labeled image

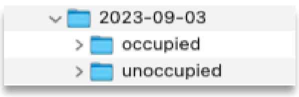

2023-09-03
occupied
unoccupied

Figure 9: dataset

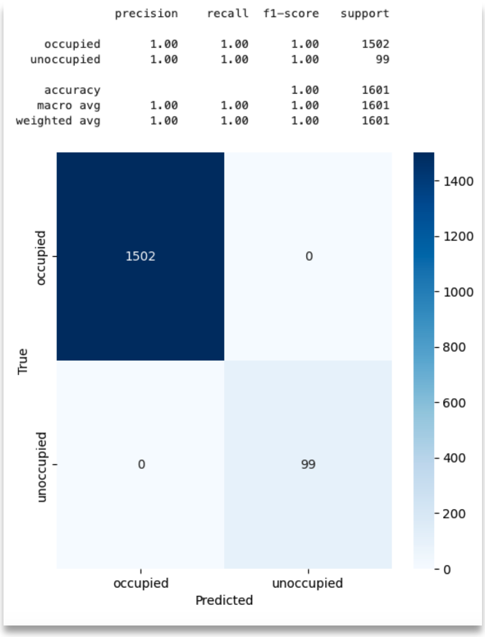
IV. Loss Function & Optimizer

- **Optimizer:** The project utilizes Adam Optimizer for adaptive learning rate with the following parameters TensorFlow RMSPropOptimizer with a momentum of 0.9, an initial learning rate of 0.001, and a learning rate decay of 0.01 every 3 epochs. A dropout rate of 0.8 and an L2 weight decay of 1e-5 are used.

V. Training

- **Infrastructure:** Conducted in the vast.ai cloud on an NVIDIA GPUs using CUDA
- **Image Processing:** resizing to 224x224 proportions, tensor conversion and the pixels values are normalized using the mean and standard deviation values [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225], respectively.
- **Load:** Images are loaded from a directory structure organized dataset using ‘ImageFolder’
- **Dataset Division:** A 90-10 split for training and validation sets.
- **Balancing:** Class weights are calculated to address class imbalance.
- WeightedRandomSampler is used to ensure a balanced distribution of classes during training.
- **Model Initiation:** The MobileNetV3 Large model “IMAGENET1K_V2” weights.
- **Training loop:** Involves 10 epochs encompassing forward/backward passes, loss computation, and updates via the Adam optimizer. Validation loss is assessed at each epoch's end, visualized through a live plot function.

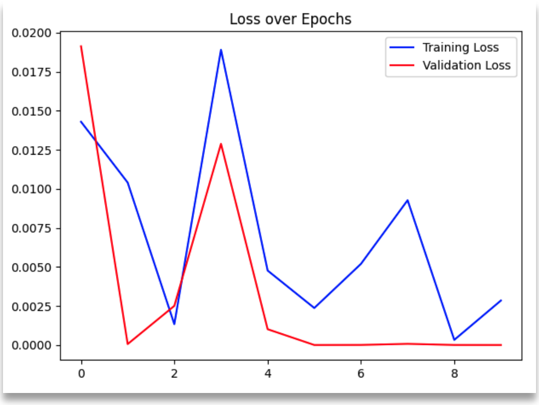
VI. Evaluation



```
              precision    recall  f1-score   support

    occupied       1.00      1.00      1.00      1502
  unoccupied       1.00      1.00      1.00        99

    accuracy                           1.00      1601
   macro avg       1.00      1.00      1.00      1601
weighted avg       1.00      1.00      1.00      1601
```



Note: Steps 3-5 take place in the vast.ai cloud

VII. Deployment

- The model is integrated into the CPPS-Server via the pytorch_model.py on-device for real-time predictions.

Classes and Files:

CPPS-Server:

app.js

Purpose: Serves as the main entry point for the server application.
Functionality: Initializes the Express server, sets up middleware like CORS and bodyParser, connects to the database, and starts the CPPS process.
Contribution: Orchestrates the startup of the server and integrates various components of the system.

appDebug.js

Purpose: Entry point for the application in debug mode.
Functionality: Similar to app.js but initiates the debug version of the CPPS process.
Contribution: Facilitates debugging and testing of the server application.

config/database.js

Purpose: Manages database connection.
Functionality: Provides a function to connect to MongoDB using Mongoose.
Contribution: Centralizes database connection logic, essential for data storage and retrieval.

config/env.js

Purpose: Manages environment variables.
Functionality: Loads environment variables using dotenv.
Contribution: Ensures secure and flexible configuration management.

debug-src/orchestra-conductor/debugCPPS.js

Purpose: Debugging script for the CPPS process.
Functionality: Executes child processes for various tasks like cropping pictures, saving them, and running predictions.
Contribution: Aids in debugging the CPPS process by providing a controlled environment for testing individual components and calibrating the blueprint.

phone-health/battery.js

Purpose: Monitors the battery status of the device.
Functionality: Executes a command to check the battery status and logs the output.
Contribution: Ensures the device Battery is charging, which is crucial for uninterrupted service.

src/capture/captureWapper.js

Purpose: Captures photos from the device camera.
Functionality: Uses child process to execute a command for taking a photo.
Contribution: Provides the raw data (images) necessary for the CPPS process.

src/data-preparation/Blueprint.js

Purpose: Manages the blueprint data for parking lots.
Functionality: Parses a JSON file to extract parking lot blueprint information.
Contribution: Essential for understanding the layout of parking lots, which is critical for the prediction process

src/data-preparation/Coordinate.js

Purpose: Represents a coordinate system for parking slots.
Functionality: Manages coordinates (x1, y1, width, height) for parking slots.
Contribution: Provides a structured way to handle spatial data for parking slots.

src/data-preparation/Slot.js

Purpose: Represents individual parking slots.
Functionality: Manages data for each parking slot, including coordinates, predictions, and image data.
Contribution: Key to the parking lot management system, as it holds detailed information about each parking slot.

src/data-preparation/Document.js

Purpose: Represents a document containing information about processed parking lot images.
Functionality: Constructs a document with parking lot images, slots, and related data.
Contribution: organizing and managing data related to each parking lot image processed.

src/data-preparation/croppedFileNames.js

Purpose: Generates names for cropped image files.
Functionality: Creates a list of filenames for cropped images based on a base name and number of images.
Contribution: Streamlines the process of managing and referencing cropped images in the system.

src/events/index.js

Purpose: Manages event handling for the CPPS process.
Functionality: Implements an event emitter for signaling the completion or error of the CPPS pipeline.
Contribution: Facilitates communication between different parts of the application, especially for handling asynchronous operations.

src/logger/logger.js

Purpose: JS-based logging functionality.
Functionality: Implements logging with various transports like file, Telegram, email, and Slack.
Contribution: Essential for monitoring, debugging, and alerting purposes.

src/logger/logger.py

Purpose: Python-based logging functionality.
Functionality: Sets up a rotating file handler for logging in Python scripts.
Contribution: Complements the JavaScript logger, ensuring comprehensive logging across the system.

src/orchestra-conductor/startCPPS.js

Purpose: Orchestrates the start of the CPPS process.
Functionality: Manages the sequence of operations like image capture, rotation, cropping, and prediction.
Contribution: Acts as the conductor for the entire CPPS process, ensuring each step is executed in order.

src/predict/pytorch_model.py

Purpose: Implements the machine learning model for predicting parking slot occupancy.
Functionality: Loads a PyTorch model, performs image transformations, and predicts occupancy.
Contribution: Central to the machine learning aspect, providing predictions on parking slot occupancy.

src/process/crop_pics.py

Purpose: Handles the cropping of parking lot images.
Functionality: Uses OpenCV for image processing and cropping based on coordinates.
Contribution: Essential for preparing images for the prediction model.

src/process/save_pic.py

Purpose: Handles saving of cropped images.
Functionality: Decodes base64 encoded images and saves them to a specified path.
Contribution: Essential for storing processed images for further analysis or record-keeping.

src/socket-server/unixDomainSocketServer.js

Purpose: Manages the Unix domain socket server.
Functionality: Sets up and handles communication via Unix domain sockets.
Contribution: Facilitates inter-process communication, crucial for coordinating various parts of the system.

src/store/db-models/parkingLots.js

Purpose: Defines the MongoDB schema for parking lots.
Functionality: Creates a schema with fields for file name, slots, and parking name.
Contribution: Essential for data persistence and retrieval related to parking lot information.

src/store/store.js

Purpose: Manages data storage operations.
Functionality: Provides functionality to store parking lot data in the database.
Contribution: Central to managing data flow to and from the database, ensuring data integrity and accessibility.

Classification-Server (used in CPPS-Server as outside service)

flask_socketio_server.py

Purpose: Serves as the core server application that integrates Flask and SocketIO for real-time communication.
Functionality: This Python script initializes a Flask application and sets up a SocketIO server. It handles incoming
'image_list' events, where it receives a list of base64-encoded images. These images are decoded, and predictions are made using the VehicleDetector class (from vehicle_detector.py). The script then emits the predictions back to the client.
Contribution: This file is crucial for real-time image processing and vehicle detection, enabling the server to Receive images, process them, and respond with predictions.

vehicle_detector.py

Purpose: Implements the vehicle detection functionality.
Functionality: The VehicleDetector class in this file loads a YOLOv4 model and performs vehicle detection. It processes input images and determines if a vehicle is present based on predefined classes of interest.
Contribution: This class is integral to the application, providing the core functionality for detecting vehicles

API

app.js

Purpose: Serves as the entry point for the Express application.
Functionality: Initializes the Express server, sets up middleware, establishes database connection, and defines routes.
Contribution: Orchestrates the overall functionality of the application, including real-time updates and request handling.

config/database.js

Purpose: Manages the database connection.
Functionality: Sets up the MongoDB connection using Mongoose, including connection URI and options.
Contribution: Ensures reliable and configurable database connectivity, vital for data storage and retrieval.

config/env.js

Purpose: Handles environment variable configuration.
Functionality: Loads environment variables from a .env file and exports them for use throughout the application.
Contribution: Centralizes configuration management, making the application adaptable to different environments.

src/authentication/authenticateHTTP.js

Purpose: Handles HTTP authentication.
Functionality: Verifies JWT tokens for HTTP requests.
Contribution: Secures HTTP endpoints by ensuring only authenticated users can access them.

src/authentication/authenticateWebSock.js

Purpose: Manages WebSocket authentication.
Functionality: Validates JWT tokens for WebSocket connections.
Contribution: Provides security for WebSocket connections, ensuring real-time data is transmitted securely.

src/controller/DocumentStorage.js

Purpose: Manages the storage and retrieval of the latest document.
Functionality: Provides functions to set and get the latest document, likely related to parking lot data.
Contribution: Key for maintaining the state of the latest parking lot data within the application.

src/logger/logger.js

Purpose: Sets up application logging.
Functionality: Configures Winston logger with custom formatting and Slack integration for real-time logging.
Contribution: Crucial for monitoring, debugging, and alerting, enhancing the maintainability of the application.

src/models/parkingLots.js

Purpose: Defines the MongoDB schema for parking lots.
Functionality: Creates a schema with fields for file name, slots, and parking name, including nested schemas for coordinates and predictions.
Contribution: Essential for data persistence and retrieval related to parking lot information, enabling structured data storage.

src/models/user.js

Purpose: Defines the user data model.
Functionality: Creates a Mongoose schema and model for user data.
Contribution: Supports user management, including authentication and token storage.

src/routers/publisher/index.js

Purpose: Manages the publisher's routing.
Functionality: Registers endpoints for publishing and welcoming users.
Contribution: Directs requests related to publishing the latest document to appropriate handlers.

src/routers/publisher/publishLatestDocument.js

Purpose: Provides an API endpoint to publish the latest document.
Functionality: Handles requests to publish the latest parking lot data.
Contribution: Enables real-time publishing of the latest parking lot data.

src/routers/publisher/welcome.js

Purpose: Offers a welcome message for the publisher route.
Functionality: Responds with a welcome message to GET requests.
Contribution: Enhances user experience by providing a friendly welcome message.

src/routers/register/index.js

Purpose: Manages the registration routing.
Functionality: Registers endpoints for user registration and welcoming.
Contribution: Facilitates user registration and authentication processes.

src/routers/register/register.js

Purpose: Handles user registration.
Functionality: Processes user registration requests and token generation.
Contribution: Enables new users to register and access the application.

src/routers/register/welcome.js

Purpose: Provides a welcome message for the registration route.
Functionality: Responds with a welcome message to GET requests on the registration route.
Contribution: Offers a friendly introduction to new users accessing the registration
route.

src/watch/changeStream.js

Purpose: Watches for changes in the MongoDB collection.
Functionality: Implements a change stream on the ParkingLots model to listen for database changes.
Contribution: Vital for real-time data processing, triggering actions upon data changes in the database.

Client (Telegram-bot-gui)

telegram-bot.py

Purpose: Powers the Telegram bot functionality.
Functionality: Handles bot interactions, manages commands, and communicates with users.
Contribution: Central component for user interaction and notification delivery.

ui.py

Purpose: Manages the user interface for the application.
Functionality: Defines layout, style, and interaction elements of the user interface.
Contribution: Enhances user experience by providing an intuitive and responsive interface.

generate_image.py

Purpose: Generates images for various purposes within the application.
Functionality: Creates and manipulates images, possibly for notifications or UI elements.
Contribution: Supports visual elements of the application, aiding in information presentation.

parking_lot.xml

Purpose: Stores configuration or data related to parking lots.
Functionality: Acts as a data source, potentially for parking lot information or settings.
Contribution: Provides essential data needed for the application's core functions.

green-indicator.png

Purpose: Represents a visual indicator, likely for an available parking space.
Functionality: Used in the UI to visually represent parking lot status.
Contribution: Enhances user understanding of parking lot availability at a glance.

red-indicator.png

Purpose: Visual indicator for an occupied or unavailable parking space.
Functionality: Used in UI to denote parking spaces that are currently occupied.
Contribution: Provides immediate visual feedback on parking lot occupancy.

working-bg-old-ver.png

Purpose: An older version of a background image for the application.
Functionality: Used as a UI background element in a previous version.
Contribution: Contributed to the aesthetic appeal of an earlier version of the application.

working-bg.png

Purpose: Current background image for the application.
Functionality: Serves as the backdrop for the application's user interface.
Contribution: Enhances the visual appeal and user experience of the application.

Arial.ttf

Purpose: Font file for the Arial typeface.
Functionality: Provides a consistent font style across the application.
Contribution: Contributes to the overall visual consistency and readability of the UI.

Pipfile & Pipfile.lock

Purpose: Manages dependencies for the Python project.
Functionality: Specifies and locks the versions of libraries required by the application.
Contribution: Ensures consistent and reproducible builds by managing package versions.

Manuals

CPPS-Server

Classification-server

Prerequisites

- **Python:** Ensure Python is installed on your machine. You can download it from the official Python website.
- **OpenCV:** This API requires OpenCV for Python. Install it using pip: pip install opencv-python.
- **Flask and Flask-SocketIO:** Required for the server. Install using pip: pip install Flask Flask-SocketIO.
- **YOLOv4 Model Files:** Download yolov4.cfg and yolov4.weights from the provided links and place them in the project directory.

Setup

Clone the Repo: Clone the GitHub repository for the project.

```
$   git clone https://github.com/parkingLotsNotifier/classification-server
$   cd classification-server
```

Starting the classification-server

Run the API: Execute the flask_socketio_server.py script to start the Flask server with SocketIO.

```
$   python flask_socketio_server.py
```

Using the classification-server

1. WebSocket Connection:

   - **URL:** Connect to the WebSocket server at ws://<server_address>:8001.
     Replace <server_address> with the actual address where the server is running.
   - **Event for Sending Images:** image_list
   - **Request Format:** Send list of base64-encoded images via the WebSocket connection using the image_list event.
   - **Description:** Use this connection to send a list of images for vehicle detection.
     The images should be base64-encoded.

2. Receiving Predictions:

   - **Event for Receiving Predictions:** `predictions`.
   - **Response Format:** The server will process each image for vehicle detection and emit a list of predictions using the `predictions` event.
   - **Description:** After processing the images, the server will send back the classification results.

API

Prerequisites

- **Node.js and npm:** Ensure you have Node.js and npm installed on your machine. You can download them from Node.js official website.
- **MongoDB:** This API requires a MongoDB database. You can either install MongoDB locally or use a cloud-based MongoDB service like MongoDB Atlas.
- **Git (Optional):** To clone the repository, you'll need Git. You can download it from Git's official site.

Setup

1. Clone the Repo:

```
$  git clone https://github.com/parkingLotsNotifier/API.git
$  cd API
```

2. *Install Dependencies:*

```
$  npm install
```

3. Environment Variables:
   - Create a .env file in the root directory of the project.
   - Add the necessary environment variables as per the config/env.js file. This includes database credentials, server port, and any other required configurations.

Starting the API

Run the API:

```
$  node app.js
```

Using the API

1. User Registration:
   - Endpoint: /auth/register
   - Method: POST
   - Body: { "username": "yourUsername", "password": "yourPassword" }
   - Description: Register a new user and receive an authentication token.
   - Response: A JWT token upon successful registration.

2. Publish Latest Document:
   - Endpoint: /api/publishLatestDocument
   - Method: POST
   - Headers: Authorization: Bearer <Your_Token>
   - Description: Publish the latest document. Requires authentication.
   - Response: Confirmation message or the latest document data

3. WebSocket Connection:
   - URL: ws://<server_address>:<port>
   - Query Parameter: ?token=<Your_Token>
   - Description: Establish a WebSocket connection for real-time updates. Use the token received during registration for authentication.
   - Usage: Establish a WebSocket connection using the provided URL.
     Authenticate the connection using the JWT token received during registration.
     Listen for messages from the server, which will include real-time updates.

Client

Telegram bot gui Manual

Prerequisites

- **Python:** Ensure you have Python installed on your system. You can download the latest version from the Python official website.
- **Pipenv:** This project uses Pipenv to manage dependencies.
  Install Pipenv by running `pip install pipenv` if you haven't installed it already.
- **Git (Optional):** Git is recommended for cloning the repository. Download it from Git's official site.

Setup

1. Clone the Repo:

   Clone the repository to your local machine with the following commands:

```
$ git clone https://github.com/parkingLotsNotifier/telegram-bot-gui.git
$ cd telegram-bot-gui
```

2. Install Dependencies:

   Inside the repository directory, install the necessary Python packages using Pipenv:

```
$ pipenv install
```

3. Set Environment Variables:

   Create a `.env` file in the root directory. Populate it with the necessary environment variables:

   TELEGRAM_TOKEN, ACCESS_TOKEN_SECRET, ALLOWED_USER_IDS, GMAIL_USERNAME,

   TELEGRAM_BOT_GMAIL_PASSWORD, MAIL_DEST2, MAIL_DEST2.

*Note:

TELEGRAM_TOKEN generated by the BotFather

ACCESS_TOKEN_SECRET generated by the API using the /auth/register end point.

ALLOWED_USER_IDS is a list of allowed telegram users ids to interact with the bot.

Running the Application

1. Activate the Virtual Environment:

   Before running the application, activate the virtual environment created by Pipenv:

```
$ pipenv shell
```

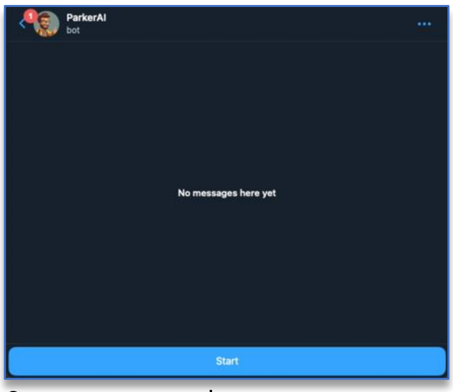
2. Run the Bot:

   Start the Telegram bot using the following command:

```
$ python telegram-bot.py
```

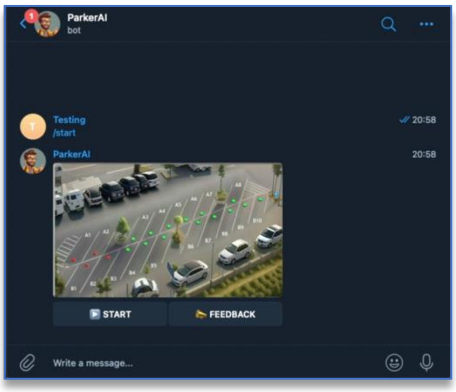
Using the Bot

1. Initialize bot:



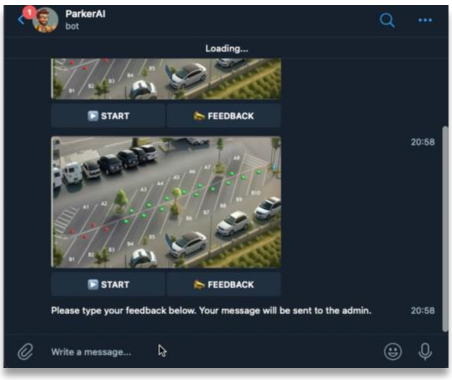
No messages here yet

2. Start request updates:



Testing
/start
ParkerAI

3. Feedback request:



Please type your feedback below. Your message will be sent to the admin.

Termux Remote access Guide

1. UPDATING AND UPGRADING PACKAGES

Before starting, ensure your packages are up-to-date:

```
$ pkg update
$ pkg upgrade
```

2. INSTALLING AND VERIFYING SSH

Refer to Termux Wiki on Remote Access for detailed information.

2.1 INSTALL OPENSSH:

```
$ pkg install openssh
```

2.2 SETTING UP PASSWORD AUTHENTICATION:

```
$ passwd
```

2.3 VERIFY OPENSSH INSTALLATION:

```
$ sshd
```

3. On Termux:

3.1 Start the SSH daemon:

```
$ sshd
```

3.2 Connect to the server (default port):

```
$ ssh user@hostname_or_ip
```

Or, if the port has changed:

```
$ ssh user@hostname_or_ip -p 8022
```

4. INSTALLING AND CONFIGURING TOR

Refer to Termux Wiki on Bypassing NAT with Tor for additional details.

4.1. INSTALL TOR AND PROXYCHAINS:

```
$ pkg install tor
$ pkg install proxychains-ng
```

4.2. SETTING UP ONION SERVICE:

4.2.1 Edit the Tor configuration file:

```
$ nano $PREFIX/etc/tor/torrc
```

4.2.2 Add the following lines at the bottom:

```
## Enable TOR SOCKS proxy SOCKSPort
127.0.0.1:9050
## Hidden Service: SSH
HiddenServiceDir /data/data/com.termux/files/home/.tor/ hidden_ssh
HiddenServicePort 22 127.0.0.1:8022
```

4.2.3 Save the file and exit.

4.3 CREATE A DIRECTORY FOR HIDDEN SERVICE:

```
$ mkdir -p ~/.tor/hidden_ssh
```

4.4 START TOR:

```
$ tor
```

4.5 RETRIEVE YOUR ONION SERVICE HOSTNAME:

```
$ cat ~/.tor/hidden_ssh/hostname
```

5. SETTING UP THE SERVER

5.1. START SSH DAEMON:

```
$ sshd
```

5.2. START TOR:

```
$ tor
```

6. CLIENT SETUP

6.1. LINUX:

6.1.1. Install Tor using your package manager.
6.1.2. Start Tor with root privileges.
6.1.3. In a new terminal, connect using:

```
$ sudo torify ssh user@your_onion_service.onion
```

6.2. WINDOWS:

6.2.1. Download the Tor expert bundle (not the browser) from Tor Project. Extract it to C:\
6.2.2. Open PowerShell as Administrator in the extracted directory and execute:

```
./tor.exe
```

6.2.3. Download PuTTY from here.
6.2.4. Follow the instructions on Tor Stack Exchange to anonymize SSH traffic.
6.2.5. Configure PuTTY:
In Session -> Host Name, enter: username@your_onion_service.onion
In Connection -> Proxy, enter: 127.0.0.1 and port 9050.
Set Proxy type to SOCKS