

1 Usługi i aplikacje Internetu Rzeczy - projekt

1.1 Kamera z czujnikiem ruchu połączona z aplikacją mobilną

Aleksander Pajorski, Jan Narożny

1.1.1 1. Raspberry Pi

1.1 Wymagane elementy

- Raspberry Pi
- Kompatybilna kamera
- Kompatybilny czujnik ruchu PIR

Raspberry Pi powinno być zaktualizowane do najnowszej dystrybucji Raspbian OS 'bookworm'. Należy wykonać

```
sudo apt update && sudo apt full-upgrade
```

aby upewnić się że biblioteki potrzebne do obsługi kamery są dostępne i aktualne.

1.2 Podłączenie kamery i czujnika Kamere podłączyć należy przy użyciu dedykowanego kabla oraz portu na płycie Pi. Kabel musi być dokładnie osadzony, częścią z kontaktami skierowany w przeciwnym kierunku niż zatrzask, a sam zatrzask równie dociśnięty. Czujnik ruchu podłączyć według poniższego schematu:

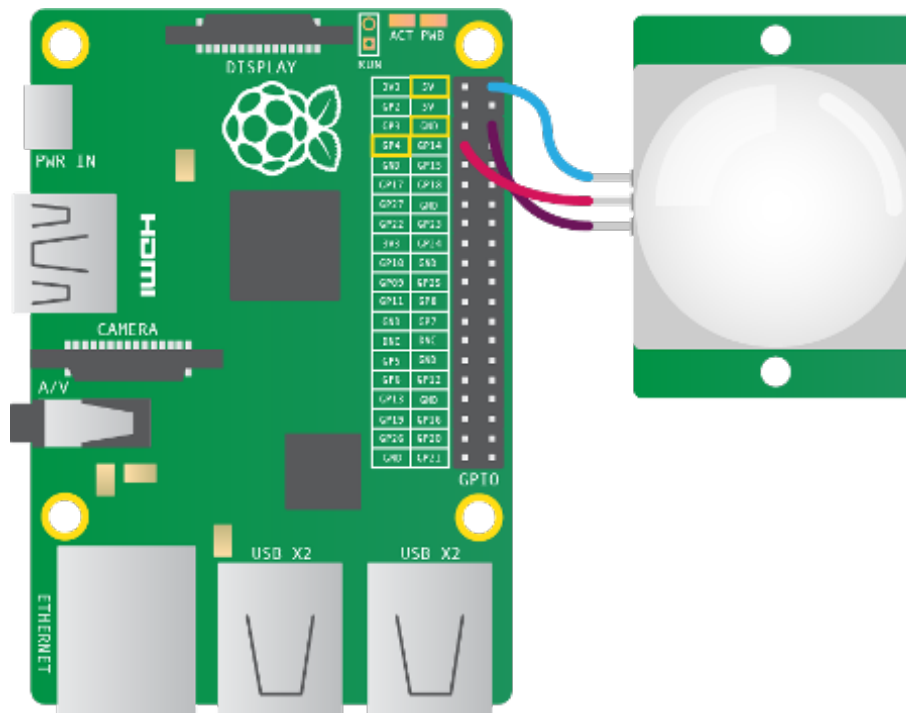


Figure 1: Schemat podłączenia czujnika ruchu

Czujnik ruchu posiada 3 piny: Vcc, Gnd, oraz Out. Powinny być one podpisane. Powyższy schemat jest poglądowy ponieważ w zależności od użytej wersji płytki ułożenie pinów GPIO może się różnić. Pin Vcc na czujniku podłączyć do pinu zasilającego 5V, pin Gnd do analogicznego pinu na płytce Pi, a Out do któregoś z pinów GPIO. W tym przypadku użyty został pin 4. Do przetestowania podłączenia czujnika ruchu:

1.3 Stworzenie python virtual environment

```
python3 -m venv venv
```

```
# windows:
```

```
venv\Scripts\activate
```

```
# macOS i linux:
```

```
source venv/scripts/activate
```

```
pip install -r requirements.txt
```

1.4 Weryfikacja podłączenia kamery i czunika Aby zweryfikować poprawne podłączenie czujnika:

```
python3 pirTest.py
```

Zakończyć ctrl+c. Następnie aby zweryfikować podłączenie kamery:

```
rpicam-still -n -o test.jpg
```

1.5 Chmura azure Zdjęcia przesyłane są do chmury azure, na którym założony został serwis Azure Blob Storage do przetrzymywania danych. Ze strony głównej Azure Portal należy wybrać “Create resource” i dodać do swojej grupy zasobów “Storage account”.

Create a storage account ...

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more about Azure storage accounts](#) >

Project details

Select the subscription in which to create the new storage account. Choose a new or existing resource group to organize and manage your storage account together with other resources.

Subscription * ▼ Azure subscription 1

Resource group * ▼ DefaultResourceGroup-CCAN [Create new](#)

Instance details

Storage account name * ⓘ

✖ The value must not be empty.

Region * ⓘ ▼ (Europe) Germany West Central [Deploy to an Azure Extended Zone](#)

Primary service ⓘ ▼ Select a primary service

Performance * ⓘ

☒ Standard: Recommended for most scenarios (general-purpose v2 account)

☐ Premium: Recommended for scenarios that require low latency.

Redundancy * ⓘ ▼ Geo-redundant storage (GRS)

☒ Make read access to data available in the event of regional unavailability.

Previous Next Review + create

Figure 2: Utwórz blob storage

W tym miejscu trzeba uzupełnić niezbędne pola: nadać nazwę, wybrać najbliższy region, primary service może zostać puste, Performance ustawić na standard a Redundancy na Locally redundant storage dla najniższych kosztów.


Po utworzeniu tego zasobu należy go otworzyć, z lewego menu wybrać opcję Containers, i kliknąć opcję dodania kontenera. Opcje zaawansowane na potrzeby tego projektu są zbędne, więc wystarczy nadać mu nazwę i wybrać poziom dostępu.

New container

Name *

Anonymous access level ⓘ

Blob (anonymous read access for blobs only) ▼

 Blobs within the container can be read by anonymous request, but container data is not available. Anonymous clients cannot enumerate the blobs within the container.

▼ Advanced

Figure 3: Container

Następnie należy przejść do “Access keys” i pamiętać o skopiowaniu “Connection string” do kodu na raspberry pi (czy jakimkolwiek innym urządzeniu, które będzie chciało uzyskać dostęp do tego Blob Storage).

storageelekpajor | Access keys

Storage account

acce

Set rotation reminder

Refresh

Give feedback

Overview

Access Control (IAM)

Security + networking

Access keys

Shared access signature

Encryption

Data management

Lifecycle management

Settings

Configuration

Access keys authenticate your applications' requests to this storage account. Keep your keys in a secure location like Azure Key Vault, and replace them often with new keys. The two keys allow you to replace one while still using the other.

Remember to update the keys with any Azure resources and apps that use this storage account.
[Learn more about managing storage account access keys](#)

Storage account name

storageelekpajor

key1

Rotate key

Last rotated: 16.11.2024 (8 days ago)

Key

.....

Show

Connection string

.....

Show

Figure 4: Connection string

1.6 Aplikacja mobilna Aplikacja mobilna napisana została przy użyciu frameworka React Native oraz Expo w języku TypeScript. PhotoGallery jest głównym komponentem aplikacji. Po jej otwarciu zdjęcia wczytywane są automatycznie, lecz nie są automatycznie odświeżane i w przypadku nadejścia nowego zdjęcia należy ręcznie odświeżyć stronę przesuwając palcem w dół do ukazania się kółka

ładowania. Obok zdjęć znajduje się data oraz godzina ich utworzenia.



Figure 5: Aplikacja mobilna

```
import React, { useEffect, useState } from 'react';
import { View, Text, FlatList, Image, ActivityIndicator, StyleSheet } from 'react-native';
import { BlobServiceClient } from '@azure/storage-blob';

const PhotoGallery = () => {
  const [photos, setPhotos] = useState<any[]>([]);
  const [loading, setLoading] = useState<boolean>(true);
```

```

const [refreshing, setRefreshing] = useState<boolean>(false);

const fetchImages = async () => {
  try {
    setLoading(true);
    const blobServiceClient = new BlobServiceClient(
      "https://name.blob.core.windows.net");
    const containerClient = blobServiceClient.getContainerClient('photos');

    const imageDetails: any[] = [];
    for await (const blob of containerClient.listBlobsFlat()) {
      if (blob.name) {
        const blockBlobClient = containerClient.getBlockBlobClient(blob.name);
        const properties = await blockBlobClient.getProperties();
        const lastModified = properties.lastModified;
        const imageUrl = `${containerClient.url}/${blob.name}`;
        imageDetails.push({ imageUrl, lastModified });
      }
    }

    const sortedImages = imageDetails.sort((a, b) => {
      if (a.lastModified && b.lastModified) {
        return b.lastModified.getTime() - a.lastModified.getTime();
      }
      return 0;
    });

    setPhotos(sortedImages);
    setLoading(false);
    setRefreshing(false);
  } catch (error) {
    console.error('Error fetching images:', error);
    setLoading(false);
    setRefreshing(false);
  }
};

useEffect(() => {
  fetchImages();
}, []);

const onRefresh = () => {
  setRefreshing(true);
  fetchImages();
};

```

```

    if (loading) {
      return (
        <View style={styles.loaderContainer}>
          <ActivityIndicator size="large" color="#aaaaaa" />
        </View>
      );
    }

    return (
      <FlatList
        style={{ backgroundColor: "#2b2b2b" }}
        data={photos}
        keyExtractor={(item, index) => index.toString()}
        renderItem={({ item }) => (
          <View style={styles.itemContainer}>
            <Image source={{ uri: item.imageUrl }} style={styles.image} />
            <Text style={styles.time}>
              {item.lastModified?.toLocaleString() || "Unknown"}
            </Text>
          </View>
        )}
        onRefresh={onRefresh}
        refreshing={refreshing}
      />
    );
  };

  const styles = StyleSheet.create({
    loaderContainer: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
    },
    itemContainer: {
      flexDirection: 'row',
      margin: 10,
      alignItems: 'center',
    },
    image: {
      width: 220,
      height: 200,
      borderRadius: 10,
    },
    time: {
      marginLeft: 10,

```

```

        fontSize: 13,
        color: 'white',
        fontWeight: '700',
    },
});

export default PhotoGallery;

```

Paczka “@azure/storage-blob” daje gotowe API do komunikacji z Azure Blob Storage. Funkcja `fetchImages()` tworzy instancję `BlobServiceClient` z podanego linku do zasobu oraz pobiera z podanego kontenera wszystkie dane. Z każdego pobranego Blob’a następnie wyciąga pola związane z URL do zdjęcia (do jego wyświetlania) oraz ostatnią modyfikacją (do wyświetlania daty i godziny jego dodania). Później zdjęcia są sortowane po dacie dodania tak, żeby jako pierwsze wyświetlały się najnowsze zdjęcia i ostatecznie przypisywana jest lista struktur `{ imageUrl, lastModified }`.

1.7 Kod Raspberry Pi Raspberry Pi obsługuje kamerę, czujnik ruchu oraz wysyłanie zdjęć do chmury skryptem Python. Podzielony jest na część konfiguracji gdzie zdefiniowany jest folder lokalny dla wykonanych zdjęć oraz parametry Blob Storage, i definicje funkcji do wykonywania i przesyłania zdjęć.

```

from azure.storage.blob import BlobServiceClient
import subprocess
from gpiozero import MotionSensor
from datetime import datetime
import sys

# Configurations
connection_string = "<connection_string>"
container_name = "<container_name>"
image_path = "tmp/"
pir = MotionSensor(4)

def take_pic(blob_name):
    output = image_path + blob_name
    try:
        subprocess.run(["sudo rpicas-still --nopreview -o " + output], check=True, shell=True)
    except subprocess.CalledProcessError as e:
        raise RuntimeError("command '{}' returned with error (code {}): {}".format(e.cmd, e.

def upload_pic(blob_name):
    src = image_path + blob_name
    try:

```



```

        client = BlobServiceClient.from_connection_string(connection_string)
        blob_client = client.get_blob_client(container=container_name, blob=blob_name)

        with open(src, "rb") as image_file:
            blob_client.upload_blob(image_file)

        print(f"Image uploaded successfully to {container_name}/{blob_name}")
    except Exception as e:
        print("Error uploading file: ", e)

def exit_gracefully():
    print("Interrupt encountered. Exiting...")
    sys.exit(0)

def main():
    blob_name = ""
    while True:
        pir.wait_for_motion()
        blob_name = "img_" + datetime.now().strftime("%H:%M:%S") + ".jpg"
        take_pic(blob_name)
        upload_pic(blob_name)
        pir.wait_for_no_motion()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt as e:
        pass
    finally:
        exit_gracefully()

```

Paczka “azure.storage.blob” zawiera funkcje potrzebne do obsługi przesyłania zdjęć do Blob Storage: `from_connection_string(connection_string)` tworzy instancję `BlobServiceClient` według danych zawartych w ‘connection_string’; `get_blob_client(container, blob)` inicjalizuje interakcję klienta z zadany ‘blob-em’, który jest wysyłany do serwera po wywołaniu `upload_blob(file)`. W tym wypadku ‘file’ zawiera lokalną ścieżkę do zdjęcia.

Paczka `subprocess` jest używana do uruchamiania polecenia odpowiedzialnego za wykonanie zdjęcia jako nowego procesu, podobnie jak w przypadku wykonania polecenia w wierszu poleceń.

Paczka “gpiozero” dedykowana jest do obsługi GPIO na Raspberry Pi, co znacznie ułatwia obsługę urządzeń takich jak czujnik ruchu poprzez dostarczenie gotowych definicji klas jak ‘`MotionSensor`’, zawierających wszelkie przydatne funkcje np. `pir.wait_for_motion()`.

W `main()` znajduje się główna pętla programu:

1. jeśli czujnik ruchu wyśle sygnał do zdefiniowanego pinu GPIO:
2. wygenerowana zostaje nazwa pliku `blob_name = "img_<czas_teraz>"`
3. wywołana zostaje funkcja `take_pic()`
4. następnie `upload_pic()`
5. program czeka na koniec sygnału z czujnika ruchu aby nie wykonał zbyt wielu zdjęć tego samego zajścia.