Interfejsy REST-owe w hobbystycznych systemach wbudowanych

Arduino + REST API = Internet Rzeczy

Obecne systemy elektroniczne, zwłaszcza urządzenia Internetu Rzeczy, korzystają ze standardowych interfejsów do komunikacji z systemami w sieciWeb. Dzięki standaryzacji systemy wbudowane mogą z łatwością łączyć się z dowolnymi usługami w Internecie, ułatwiając integrację systemów współczesnego świata.

Standardowe interfejsy używane są w elektronicznych systemach cyfrowych niemalże od zawsze. Każdy elektronik, który kiedyś miał do czynienia z mikrokontrolerami, wie, czym jest UART, SPI czy I²C. Dzięki standaryzacji w przemyśle elektronicznym układy różnych producentów mogą, bez zbytnich komplikacji, komunikować się ze sobą.

Obecnie coraz większe znaczenie w świecie elektroniki mają urządzenia komunikujące się z Internetem czy działające w sieci lokalnej. Do niedawna Internet był zarezerwowany dla komputerów, ale wprowadzenie koncepcji Internetu Rzeczy (IoT – *Internet of Things*) podłączyło do sieci szerokie spektrum innych urządzeń. Rzeczy ewoluowały w wyniku konwergencji wielu technologii: analizy danych w czasie rzeczywistym (zwłaszcza na niewielkich systemach wbudowanych), uczenia maszynowego, rozwoju czujników towarów i ewolucji samych systemów wbudowanych.

Tradycyjne dziedziny systemów wbudowanych – bezprzewodowe sieci czujników, systemy sterowania, automatyka (szczególnie automatyka domowa i budynkowa) i inne, przyczyniają się do rozwoju Internetu Rzeczy. Na rynku konsumenckim technologia IoT jest obecnie synonimem produktów związanych z koncepcją inteligentnego domu, w tym urządzeń takich jak inteligentne oprawy oświetleniowe, termostaty, systemy i kamery bezpieczeństwa do domu oraz inne urządzenia automatyki domowej, które obsługują jeden wspólny ekosystem wymiany danych i można nimi sterować za pomocą innych urządzeń, mogących komunikować się z tym ekosystemem, takich jak smartfony czy inteligentne głośniki. Podstawą do konstrukcji tego rodzaju standardowych interfejsów komunikacyjnych są właśnie REST API. Ich rozpowszechnienie w elektronice sprawiło, że podłączenie nawet niewielkiego mikrokontrolera do sieci Web jest trywialnie proste.

Co to jest REST?

REST API (znany również jako RESTful API lub API REST-owe) to interfejs programowania aplikacji (API lub internetowy interfejs API), który jest zgodny z ograniczeniami stylu architektonicznego REST i umożliwia interakcję z usługami sieciowymi, które również są RESTful.

API to zbiór ścisłych reguł, które opisują sposób komunikacji programów i podprogramów ze sobą. Jest to przede wszystkim zestaw wytycznych, dotyczących przebiegu interakcji pomiędzy komponentami w programie lub programach. Sama implementacja API to zestaw funkcji i protokołów, które wykorzystywane są do komunikacji pomiędzy tymi systemami. Dobre API ułatwia budowę niezawodnego i prostego oprogramowania. Dzięki użyciu gotowych, predefiniowanych API programista musi jedynie łączyć ze sobą bloki w ustalonej konwencji. Ułatwia to łączenie ze sobą wielu subsystemów, często tworzonych przez różnych programistów czy nawet różne firmy.

API to zestaw definicji i protokołów służących do tworzenia i integracji oprogramowania. Czasami nazywa się to „umową" między dostawcą informacji a jej użytkownikiem. Umowa taka określa treść wymaganą od konsumenta (treść wezwania) i treść wymaganą przez producenta/dostawcę informacji (treść odpowiedź). Na przykład ideowy projekt API dla usługi pogodowej może określać, że użytkownik podaje kod pocztowy, a dostawca informacji odpowiada dwuczęściową odpowiedzią, z których pierwsza to górny zakres temperatur, a drugi to jej dolny zakres. Innymi słowy, jeśli chcemy współdziałać z komputerem lub innym systemem w celu pobrania z niego informacji lub wykonania jakiejś funkcji, interfejs API pomaga przekazać w standardowy sposób polecenie lub informacje, które pozwalają w pełni zrozumieć i spełnić nasze żądanie.

Można myśleć o API jako o pośredniku między użytkownikami lub klientami a zasobami lub usługami internetowymi, do których

dostęp chcą uzyskać. Jest to również sposób, w jaki organizacja może udostępniać swoje zasoby i informacje przy zachowaniu bezpieczeństwa, kontroli i uwierzytelniania – określając, kto ma do czego dostęp. Kolejną zaletą wykorzystania interfejsu API jest to, że tworząc program, który korzysta z jego zasobów, nie trzeba znać specyfiki ich buforowania – w jaki sposób dany zasób jest pobierany lub skąd pochodzi.

Listing 1. Przykładowy komunikat w formacie JSON

```
{
    „Czasopismo”:„Elektronika Praktyczna”
    „Numer”:5,
    „Spis Treści”:[
        {„Autor”:„Jan Kowalski”, „Tytuł”:”Przykładowy artykuł”},
        {„Autor”:”Nikodem Czechowski”, ”Tytuł”:”Interfejsy RESTowe”},
        {„Autor”:”Lorem Ipsum”, „Tytuł”:”Dolor sit amet, consectetur adipiscing elit”}
    ]
}
```

REST oznacza reprezentacyjny transfer stanu. Został opracowany przez informatyka Roya Fieldinga. REST to zestaw ograniczeń architektonicznych, a nie protokół czy standard. Programiści API mogą implementować REST na różne sposoby.

Gdy żądanie klienta jest wysyłane za pośrednictwem RESTful API, przekazuje reprezentację stanu zasobu do klienta lub punktu końcowego. Te informacje lub reprezentacje są dostarczane w jednym z kilku formatów za pośrednictwem protokołu HTTP: JSON (*JavaScript Object Notation*), HTML, XLT, Python, PHP lub jako zwykły tekst. Format JSON jest najpopularniejszym używanym formatem reprezentacji stanu, ponieważ pomimo swojej nazwy jest niezależny od języka i jest czytelny zarówno dla ludzi, jak i maszyn.

System Notacji JavaScript JSON to lekki format do przechowywania i transportu danych. Tak jak opisano powyżej, format JSON jest często używany, gdy dane są wysyłane z serwera na stronę internetową lub ogólnie w sieci. Wynika to między innymi z jego ogromnej czytelności, co ułatwia debugging przy przesyłaniu JSON-ów w formacie tekstowym do sieci Web. JSON jest samoopisujący, co oznacza, że w samej paczce danych znajdują się informacje na temat zawartości poszczególnych jej części. Wszystko stanie się jasne, jeśli spojrzymy na **listing 1**, gdzie pokazano przykładowy komunikat JSON. Komunikat ten w dosyć skrajnie uproszczony i ideowy sposób opisuje ten numer czasopisma „Elektronika Praktyczna”. JSON jest zasadniczo tablicą asocjacyjną (słownikiem, czyli zestawem par klucz-wartość). Kluczami są zmienne typu String, znajdujące się w cudzysłowie, a wartościami mogą być zmienne typu String, liczby (typu Double, czyli zmiennoprzecinkowe liczby 64-bitowe) lub stałe *true*, *false* lub *null*. Elementami takiej tablicy mogą być także kolejne, zagnieżdżone tablice, złożone z takiego samego formatu elementów. Daje to ogromne możliwości przechowywania i przesyłania niemalże dowolnych danych.

Kolejnym aspektem, o którym trzeba wiedzieć, tworząc system RESTful, są nagłówki i parametry w metodach obsługujących metody HTTP w REST-owym API. Zawierają ważne informacje o identyfikatorach dotyczących metadanych żądania, autoryzacji, jednolitego identyfikatora zasobu (URI), pamięci podręcznej, plikach cookie i innych. Istnieją nagłówki żądań i nagłówki odpowiedzi, każdy z własnymi zestawami informacji o połączeniu HTTP i kodami stanu.

Aby API zostało uznane za REST-owe, musi spełniać następujące kryteria:

- Architektura klient-serwer składająca się z klientów, serwerów i zasobów z żądaniami zarządzanymi poprzez HTTP;
- Bezstanowa komunikacja klient-serwer, co oznacza, że żadne informacje o kliencie nie są przechowywane między żądaniami, a każde żądanie jest oddzielne i niepołączone z poprzednimi/kolejnymi;
- Dane przesyłane są z możliwością buforowania, która usprawnia interakcje klient-serwer.
- Jednolity interfejs między komponentami, dzięki czemu informacje są przesyłane w standardowej formie. Wymaga to:
  - żądane zasoby są możliwe do zidentyfikowania i oddzielone od reprezentacji przesłanych klientowi,
  - klient może manipulować zasobami za pośrednictwem otrzymanej reprezentacji, ponieważ reprezentacja zawiera wystarczającą ilość informacji, aby to zrobić,
  - komunikaty są samoopisane – zwracane do klienta elementy zawierają wystarczającą ilość informacji, aby opisać, jak klient powinien je przetwarzać,
  - dostępne są hiperteksty/hipermedia, co oznacza, że po uzyskaniu dostępu do zasobu klient powinien mieć możliwość korzystania z hiperłączy, aby znaleźć wszystkie inne aktualnie dostępne działania, które może wykonać;
- System warstwowy, który organizuje każdy typ serwera (odpowiedzialny za bezpieczeństwo, równoważenie obciążenia itp.), obejmuje pobieranie żądanych informacji w hierarchii, niewidocznej dla klienta;
- Kod na żądanie (opcjonalnie): możliwość wysyłania wykonywalnego kodu z serwera do klienta (na żądanie), rozszerzająca jego funkcjonalność.

Chociaż interfejs REST-owy spełnia te kryteria, nadal jest uważany za łatwiejszy w użyciu niż protokoły takie jak np. SOAP (*Simple Object Access Protocol*), który ma określone wymagania, takie jak obsługa wiadomości XML oraz wbudowane zabezpieczenia i zgodność transakcji, które sprawiają, że jest wolniejszy i cięższy obliczeniowo po stronie klienta i serwera, co utrudnia jego stosowanie na kompaktowych i energooszczędnych mikrokontrolerach. Natomiast REST to zestaw wskazówek, które można zaimplementować w razie potrzeby, dzięki czemu interfejsy RESTful są szybsze i lżejsze, a także charakteryzują się zwiększoną skalowalnością – są idealne do tworzenia aplikacji Internetu Rzeczy (IoT) i aplikacji mobilnych.

Dlaczego API REST-owe jest tak ważne w systemach webowych?

REST to styl projektowania i radzenia sobie z komunikacją, który jest bardzo rozpowszechniony w systemach webowych. Stał się on obecnie standardem dla aplikacji działających w Internecie. Opisaliśmy już jego charakterystyki, wynika z nich wiele zalet. W dużej mierze sprowadza się to do optymalizacji – wykorzystanie REST jest faworyzowane w stosunku do SOAP, ponieważ REST potrzebuje mniejszej przepustowości i pozwala na ograniczenie transferu w sieci. Kolejną zaletą REST API jest to, że jest ono przyjazne dla użytkownika i łatwe do zrozumienia dla programistów. Tworzenie tego rodzaju API jest łatwiejsze niż np. SOAP, gdyż podejście REST-owe koncentruje się na danych.

Ponieważ REST używa standardowych zapytań HTTP, interfejsy API do weryfikacji danych i weryfikacji adresu są łatwe do zrozumienia i oprogramowania dla projektantów aplikacji. Co więcej, architektury RESTful ułatwiają generowanie danych wyjściowych w bardziej elastycznych formatach danych, takich jak JSON, zamiast np. formatu XML, którego wymaga SOAP. Kluczowe założenia REST obejmują wyodrębnienie interfejsu API na prawidłowe zasoby. Te zasoby są kontrolowane za pomocą zapytań HTTP, w takim przypadku, gdy rodzaj zapytania (GET, POST, PUT, PATCH i DELETE) ma określone znaczenie.

Stopień sprzężenia między serwerem a klientem SOAP jest bardzo duży. W REST jest inaczej. Klient tego rodzaju bardziej przypomina przeglądarkę. Jest to typowy klient, który wie, jak używać protokołu HTTP i standardowych metod, a aplikacja musi mieścić się w ramach, które to narzuca. Dodatkowo klient RESTful może korzystać z usługi REST bez znajomości danego API, z wyjątkiem punktu wejścia i typu interfejsu. W przypadku SOAP klient potrzebuje wcześniejszej wiedzy

na temat wszystkiego, z czego będzie korzystał, inaczej nawet nie rozpocznie interakcji.

W REST każde żądanie klienta skierowane do serwera wymaga pełnej reprezentacji jego stanu. Serwer musi być w stanie w pełni zrozumieć żądanie klienta bez korzystania z jakiegokolwiek kontekstu lub stanu sesji. Cały stan musi być zachowany na kliencie. W SOAP nie jest to wymagane, a serwer może przechowywać zmienne sesji.

Finalnie – ogromną zaletą podejścia REST-owego jest łatwość testowania API. Istnieją dwie podstawowe sytuacje, w których testuje się REST API online. Po pierwsze po opracowaniu API trzeba uzyskać usługę sieci Web zgodną z REST i upewnić się, że działa zgodnie z założeniami. Po drugie, podczas opracowywania aplikacji, która ma korzystać z usługi w sieci Web, konieczne jest zbadanie dostępu do tej usługi. Ogromną zaletą REST jest to, ż łatwo znaleźć błędy logiczne w kodzie, ponieważ komunikaty JSON są bez problemu zrozumiałe dla człowieka i na żadnym etapie komunikacji z API nie mają one sztywnego formatu.

Implementacje REST – podstawowe zasady

Istnieje kilka sprawdzonych zasad, które mogą pomóc w zaprojektowaniu odpowiedniego RESTful API. Są one niezależne od tego, czy programujemy usługę w sieci Web, program na komputerze PC, czy też firmware dla systemu wbudowanego.

1. W nazwach należy używać rzeczowników zamiast czasowników. Używaj rzeczowników dla każdego zapytania, np. GET, POST, PUT, DELETE. Rekomenduje się używanie tylko rzeczowników w liczbie mnogiej;
2. Nie używaj metody GET do zmiany stanu. Generalnie metoda ta powinna być używana do uzyskania stanu (informacji);
3. Metoda POST powinna być używana do tworzenia. Nie należy jej używać do aktualizowania ani pobierania zasobów. Jeśli identyfikator URI nigdy wcześniej nie istniał, a zamierzasz go utworzyć i sprawić, by zawierał jakieś dane, użyj POST;
4. Do aktualizacji stanu należy używać metody HTTP PUT. Aktualizacja to zastąpienie istniejącego już zbioru danych innym, nowym. Identyfikator URI powinien już istnieć wcześniej;
5. Do usuwania należy używać protokołu DELETE;
6. Użycie GET nie powinno nic tworzyć ani mieć innych efektów, jak tylko uzyskanie informacji;
7. Metody POST nie należy używać do masowej aktualizacji danych;
8. Jeśli na jakieś URI da się wykonać POST, to należy także umożliwić wykorzystanie metody GET, która może zwrócić zapisane dane. Jeśli dane są zbiorem, to koniecznie trzeba stworzyć możliwość zwrócenia pojedynczej wartości z kolekcji, indeksując ją. Tak samo powinna działać metoda DELETE – umożliwiać skasowanie całej kolekcji lub tylko jednego jej elementu;
9. Należy korzystać z kodów HTTP, zwracanych przez interfejs, oto niektóre z nich:
   - **200**: Zapytanie poprawne – jest to kod, którego na ogół się spodziewamy;
   - **201**: Zapytanie poprawne, utworzono obiekt – kod, który potwierdza wykonanie zapytania POST;
   - **204**: Zapytanie poprawne, nie ma obiektu – kod potwierdzający wykonanie DELETE;
   - **400**: Serwer nie zrozumiał otrzymanego zapytania;
   - **401**: Błąd autoryzacji, należy najpierw się uwierzytelnić (np. zalogować);
   - **403**: Brak uprawnień – zalogowano się, ale nie daje to uprawnień do danego obiektu;
   - **404**: Nie ma obiektu – nie można znaleźć go na serwerze;
   - **410**: Zasób oznaczony, jako skasowany;
   - **418**: Serwer jest czajniczkiem i nie może parzyć kawy (tak! To realnie zdefiniowany w dokumentacji błąd – pokłosie żartu na prima aprilis twórców http);
   - **451**: Niedostępne z powodów prawnych.

Jak zaimplementować REST API w systemach wbudowanych?

Istnieje wiele sposobów na zastosowanie REST-owego interfejsu aplikacji w systemie opartym na mikrokontrolerze. W zależności od czasu, jaki chcemy poświęcić na implementację, można skorzystać z gotowych, kompletnych bibliotek oferujących kompletne REST-owe API w chmurze (jak na przykład aREST, którą można znaleźć na repozytorium autora (Marco Schwartz) na portalu GitHub – https://bit.ly/3t9LqYt), jak i bibliotek, które pozwalają na zestawienie tego rodzaju API z poszczególnych elementów, tzn. serwowania JSON-ów, obsługi zapytań http itd.

Najprostszym sposobem na implementację interfejsu REST-owego w systemie wbudowanym, jest użycie Arduino wraz z odpowiednią biblioteką dla serwera HTTP. Jeśli serwer uzupełnimy biblioteką zdolną do formatowana danych w postać JSON, uzyskamy kompletne API RESTful. Jeśli chodzi o platformę sprzętową dobrym wyborem są układy ESP8266 lub ESP32 – te SoM mają wbudowane interfejsy bezprzewodowe (Wi-Fi oraz Bluetooth), co istotnie upraszcza łączenie ich z siecią Web.

Implementacja RESTful API na układach Espressif

W poniższym przykładzie zaprezentujemy krok po kroku, jak można uruchomić REST-owe API na module z układem ES8266, korzystając z prostych bibliotek.

Serwer http

W pierwszej kolejności konieczne jest uruchomienie serwera webowego. W tym celu należy zaimportować odpowiednią bibliotekę:

```
#include <ESP8266WebServer.h>
```

Następnie należy ją skonfigurować. Kluczowym parametrem jest port, na którym uruchomi się serwer http. Aby ustawić port 8080 i przygotować serwer, musimy dodać dwie linie kodu. Port 8080 to domyślny port serwera www, dzięki czemu umożliwia korzystanie z przeglądarki:

```
#define HTTP_REST_PORT 8080
ESP8266WebServer httpRestServer(HTTP_REST_PORT);
```

W bloku *setup()* kodu Arduino wykonujemy dwie funkcje, które uruchomią routing (dzięki czemu każde zapytanie trafi w odpowiednie miejsce w programie) oraz uruchomi sam serwer:

```
restServerRouting();
httpRestServer.begin();
```

Finalnie w pętli programu (w sekcji *loop()*) należy umieścić funkcję, która obsługuje zapytania klientów, przesyłane do serwera http:

```
httpRestServer.handleClient();
```

Teraz pozostaje nam tylko stworzyć funkcje, które obsługują poszczególne zapytania i zwracają dane w formacie JSON.

Obsługa zapytania GET

Obsługa zapytań http jest bardzo prosta. W specjalnej funkcji *restServerRouting()* definiujemy, na jakie zapytania, pod jakimi ścieżkami serwer odpowiada (**listing 2**). Kluczowym obiektem jest *server*. Metoda *server.on* ustala odpowiedź na zapytanie. Weźmy na przykład linię:

```
server.on(F("/helloWorld"), HTTP_GET, getHelloWord);
```

Jako parametry podaje się tak zwany endpoint, czyli ścieżkę, pod jaką wysłane musi być zapytanie (w tym wypadku jest to */helloWorld*), rodzaj zapytania (HTTP_GET) oraz odpowiedź – funkcja *getHelloWord*, którą trzeba osobno zdefiniować. Funkcja ta na ogół zwraca

Listing 2. W specjalnej funkcji restServerRouting() jest definiowane to, na jakie zapytania, pod jakimi ścieżkami serwer odpowiada

```
void restServerRouting() {
    server.on("/", HTTP_GET, []() {
        server.send(200, F("text/html"),
        F("Welcome to the REST Web Server"));
    });
    server.on(F("/helloWorld"), HTTP_GET, getHelloWord);
}
```

odpowiedź w formacie JSON, korzystając z metody *server.send()*:

```
server.send(200, “text/json”,
“{\”name\”: \”Hello world\”}”);
```

Argumenty podawane do zapytania

Zapytanie GET używane jest zwykle do pobierania danych, takich jak np. lista elementów lub element, aby uszczegółowić, jakie elementy są konkretnie potrzebne. Możliwe jest dodanie argumentów do zapytania w postaci par klucz-wartość w adresie URL. Parametry te przekazywane są w adresie po znaku „?” i oddzielone są znakiem „&”. Na przykład URL zapytania wygląda tak:

```
/settings?SignalStrength=true&chipInfo=true&
freeHeap=true.
```

Tego rodzaju zapytanie obsługiwane jest w następujący sposób. W pierwszej kolejności metoda *server.on()*:

```
server.on(F(“/settings”), HTTP_GET, getSettings);
```

korzysta z funkcji getSettings(), której kod został pokazany na **listingu 3**. W ramach tej funkcji uruchamiana jest metoda *server.arg()*, która jako parametr przyjmuje nazwę klucza, którego wartość chcemy poznać (z podanych na endpoint wartości). Funkcja zwraca wartość dla danego klucza lub *False*, jeśli pośród podanych parametrów nie ma takiej wartości.

Listing 3. Kod funkcji getSettings(), z której w pierwszej kolejności korzysta metoda server.on()

```
void getSettings() {
    String response = "{";
    response+= "\"ip\": \""+WiFi.localIP().toString()+"\"";
    response+= ",\"gw\": \""+WiFi.gatewayIP().toString()+"\"";
    response+= ",\"nm\": \""+WiFi.subnetMask().toString()+"\"";
    if (server.arg("signalStrength")== "true"){
        response+= ",\"signalStrengh\": \""+String(WiFi.RSSI())+"\"";
    }
    if (server.arg("chipInfo")== "true"){
        response+= ",\"chipId\": \""+String(ESP.getChipId())+"\"";
        response+=",\"flashChipId\":\""+String(ESP.getFlashChipId())+"\"";
        response+= ",\"flashChipSize\": \""+String(ESP.getFlashChipSize())+"\"";
        response+= ",\"flashChipRealSize\": \""+String(ESP.getFlashChipRealSize())+"\"";
    }
    if (server.arg("freeHeap")== "true"){
        response+= ",\"freeHeap\":\""+String(ESP.getFreeHeap())+"\"";
    }
    response+="}";
    server.send(200, "text/json", response);
}
```

Listing 4. Funkcja serializeJson() konwertuje zmienną typu DynamicJsonDocument do typu String, który można wysłać przez server.send()

```
void getHelloWord() {
      DynamicJsonDocument doc(512);
      doc["name"] = "Hello world";
      String buf;
      serializeJson(doc, buf);
      server.send(200, "application/json", buf);
}
```

Listing 5. Funkcja setRoom, która zwróci nam zawartość danych, wysłanych do serwera

```
void setRoom() {
    String postBody = server.arg("plain");
    DynamicJsonDocument doc(512);
    DeserializationError error = deserializeJson(doc, postBody);
    if (error) {
        String msg = error.c_str();
        server.send(400, F("text/html"),
                "Error in parsin json body! <br>" + msg);
    } else {
        JsonObject postObj = doc.as<JsonObject>();
        if (server.method() == HTTP_POST) {
            if (postObj.containsKey("name") && postObj.containsKey("type")) {
                // Tutaj magazynujemy bądź analizujemy dane
                DynamicJsonDocument doc(512);
                doc["status"] = "OK";
                String buf;
                serializeJson(doc, buf);
                server.send(201, F("application/json"), buf);
            }else {
                DynamicJsonDocument doc(512);
                doc["status"] = "KO";
                doc["message"] = F("No data found, or incorrect!");
                String buf;
                serializeJson(doc, buf);
                server.send(400, F("application/json"), buf);
            }
        }
    }
}
```

Formatowanie JSON

Większość informacji zwracanych przez serwer klientom REST-owego API jest podawana w omawianym powyżej formacie JSON. Nic nie stoi oczywiście na przeszkodzie, aby generować go na piechotę. Można jednak skorzystać z odpowiednich bibliotek, żeby uprościć sobie generowanie komunikatów w formacie JSON.

Aby skorzystać z biblioteki w środowisku Arduino, musimy, w pierwszej kolejności, zaimportować ją do kodu:

```
#include <ArduinoJson.h>
```

Następnie korzystamy z funkcji *serializeJson()*, która konwertuje zmienną typu *DynamicJsonDocument* do typu String, który można wysłać przez *server.send()*, jak w pokazanym przypadku funkcji *getHelloWorld()*, która zwraca wartość dla klucza *name*. (**listing 4**).

Obsługa pozostałych zapytań

Oprócz zapytania GET do serwera REST może dotrzeć wiele innych zapytań. Szczególnie istotnym z nich jest POST – pozwala ono klientowi wysłać dane do serwera (na ogół w formacie JSON). Zdefiniujmy sobie funkcję do obsługi zapytania:

```
server.on(F("/setRoom"), HTTP_POST, setRoom);
```

Następnie w funkcji *setRoom*, która je obsłuży, zawrzyjmy odpowiednią funkcję *server.arg(„plain”)*, która zwróci nam zawartość danych, wysłanych do serwera. Następnie, korzystając z funkcji z biblioteki ArduinoJson.h, odczytajmy zawarte tam informacje (**listing 5**).

Podsumowanie

API RESTful jest podstawowym narzędziem do tworzenia usług w sieci Web. Pozwala na wymianę danych w czytelny i łatwy sposób. Dzięki zastosowaniu prostych do zrozumienia dla programisty słów cały interfejs jest czytelny, bez nadmiernej dokumentacji. Warto pamiętać, aby nazwami endpointów były rzeczowniki. Na przykład, w powyżej prezentowanym kodzie endpoint nazwano *setRoom*, co jest technicznie poprawne, ale błędne logicznie. Jeśli nazwiemy endpoint po prostu *room*, to wtedy wykorzystać możemy różne zapytania do konstrukcji logicznych systemów, na przykład:

- POST, aby dodać nowy pokój;
- GET, aby uzyskać informacje na temat pokoju;
- DELETE, aby usunąć pokój;
- I tak dalej...

Zachowując wszystkie opisane w artykule podstawowe rekomendacje, można zbudować łatwy w implementacji interfejs do wymiany danych w dowolnej sieci. Idealnie nadaje się to do tworzenia webowych API dla systemów Internetu Rzeczy, korzystających z modułów ekosystemu Arduino.

Nikodem Czechowski, EP

Bibliografia:

- https://bit.ly/3eegl1x
- https://bit.ly/33aIJLv
- https://bit.ly/3gYN7VW
- https://red.ht/3uf6nT5
- https://bit.ly/3xLSs95
- https://bit.ly/3gXGgML
- https://bit.ly/2QTWHPg
- https://mzl.la/2QPo5y0
- https://bit.ly/3t9LqYt
- http://arest.io/
- https://bit.ly/2QJxNSG/
- https://bit.ly/3gZJxuS
- https://bit.ly/3ug28qp