Programowanie aplikacji mobilnych (2)

Sterowanie przez Ethernet oraz wykorzystanie pamięci urządzenia

W drugiej części kursu programowania aplikacji mobilnych dla elektroników pokażemy jak wykonać przykładową aplikację, która pozwoli nam na komunikowanie się z innymi urządzeniami przez Ethernet lub połączenie sieci telefonii komórkowej. Zademonstrujemy też prosty sposób zapisywania w telefonie danych gromadzonych przez aplikację, aby można było przechowywać np. ustawienia bądź jakieś dane historyczne. Zaczniemy jednak od pokazania, jak w ogóle zabierać się do programowania, gdy mamy już zainstalowane wszystkie narzędzia deweloperskie.

Platforma Cordova, którą zainstalowaliśmy w poprzedniej części kursu, zawiera nie tylko funkcje potrzebne do wyświetlania dowolnie skomponowanych tekstów i grafik na ekranie smartfonu czy tabletu lub reagowania na dane wprowadzane dotykowo czy za pomocą klawiatury. Obejmuje też funkcje do komunikacji aplikacji z Internetem czy dowolną siecią Ethernet (w praktyce Wi-Fi), do której jest dołączone urządzenie mobilne. To już całkiem dużo i w wielu wypadkach wystarczy do realizacji prostego, bezprzewodowego kontrolera urządzenia elektronicznego, jednak czasem będą potrzebne dodatkowe funkcje umożliwiające obsługę innych komponentów

Tabela 1. Oficjalne pluginy platformy Cordova, opracowane i udostępnione przez jej twórców

Nazwa pluginu	PhoneGap Build	Opis
org.apache.cordova.battery-status	jest	Monitorowanie poziomu baterii
org.apache.cordova.camera	jest	Wykonywanie zdjęć wbudowaną kamerą i wybieranie plików graficznych spośród zapisanych w bibliotece wewnętrznej urządzenia
org.apache.cordova.console	jest	Rozszerzenie funkcji przesyłania informacji do konsoli systemowej dla niektórych platform mobilnych
org.apache.cordova.contacts	jest	Dostęp do listy kontaktów zapisanych w telefonie
org.apache.cordova.device	jest	Dostęp do informacji o sprzęcie, na którym uruchomiona jest aplikacja
org.apache.cordova.device-motion	jest	Dostęp do akcelerometru urządzenia
org.apache.cordova.device-orientation	brak	Dostęp do kompasu urządzenia
org.apache.cordova.dialogs	jest	Obsługa systemowych okienek dialogowych urządzenia
org.apache.cordova.file	jest	Dostęp do systemu plików urządzenia, ograniczony do typowo używanych folderów
org.apache.cordova.file-system-roots	jest	Dodatkowy dostęp do niektórych systemowych zasobów dyskowych, takich jak karta SD, katalogi cache i cały system plików
org.apache.cordova.file-transfer	jest	Zestaw funkcji umożliwiający przesyłanie przez Internet/Ethernet plików na zewnętrzne serwery i pobieranie ich do wybranych katalogów w urządzeniu
org.apache.cordova.geolocation	jest	Dostęp do koordynatów nawigacji satelitarnej, np. z wbudowanego odbiornika GPS
org.apache.cordova.globalization	jest	Przydatny zestaw funkcji ułatwiających sposób prezentacji informacji, zgodnie ze standardami i zwyczajami obowiązującymi w kraju użytkownika
org.apache.cordova.inappbrowser	jest	Zestaw funkcji pozwalający na wykorzystanie w aplikacji zewnętrznych stron internetowych, ograniczający ich uprawnienia. Cordova pozwala standardowo na prezentacje w aplikacji dowolnych stron internetowych, ale dopiero użycie tego pluginu odcina im dostęp do zasobów aplikacji i smartfonu
org.apache.cordova.media	jest	Zestaw funkcji umożliwiający nagrywanie i odtwarzanie dźwięku
org.apache.cordova.media-capture	jest	Dostęp do różnych funkcji związanych z rejestracją obrazów, dźwięków i nagrań wideo
org.apache.cordova.network-information	jest	Dostęp do informacji o stanie połączeń komórkowych i Wi-Fi urządzenia
org.apache.cordova.plugin.softkeyboard	brak	Funkcja pokazywania klawiatury ekranowej; tylko dla Androida
org.apache.cordova.speech.speechsynthesis	brak	Obsługa systemowego syntezatora mowy
org.apache.cordova.splashscreen	jest	Funkcje umożliwiające pokazywanie i ukrywanie ekranu powitalnego aplikacji
org.apache.cordova.statusbar	jest	Obsługa parametrów (pokazywanie, ukrywanie, styl, kolor) systemowego paska statusu urządzenia
org.apache.cordova.vibration	jest	Obsługa wibracji urządzenia

Tabela 2. Wybrane, ciekawe pluginy do Cordovy, które mogą być przydatne dla elektroników

Nazwa pluginu	PhoneGap Build	Opis
com.blackberry.app	brak	Zestaw funkcji umożliwiających obsługę zdarzeń typowych dla systemu BlackBerry 10, w tym obsługę klawiatury ekranowej
com.chariotsolutions.nfc.plugin	jest	Obsługa NFC (zapis i odczyt znaczników oraz przesyłanie wiadomości). Tylko dla Androida, Windows Phone 8 oraz BlackBerry 10 i BlackBerry 7
com.eficid.cordova.plugin.rfidscanner	jest	Ciekawy plugin, umożliwiający obsługę skanerów RFID firmy MTI, pracujących w zakresie UHF. Uwaga! Wtyczka nie jest dostępna w oficjalnym repozytorium Cordovy
com.evothings.ble	jest	Alternatywny plugin do obsługi Bluetootha. Pracuje tylko i wyłącznie z Bluetooth Low Energy i ma inny zestaw funkcji i możliwości, niż popularny com.megster.cordova.bluetoothserial
com.google.cordova.admob	brak	Interfejs umożliwiający proste wyświetlanie reklam Google z sieci AdMob w aplikacji mobilnej. Prosty sposób na zarobienie na bezpłatnej aplikacji
com.google.playservices	brak	Obsługa sklepu GooglePlay, np. celem instalacji aplikacji potrzebnych do działania tworzonego programu. Oczywiście, tylko dla Androida
com.ionic.keyboard	jest	Bardziej rozbudowana obsługa klawiatury ekranowej niż w standardowym pluginie org.apache.cordova.plugin.softkeyboard. Działa na Androidzie i na iOS
com.jsmobile.plugins.sms	brak	Jeden z pluginów do obsługi SMSów
com.megster.cordova.ble	jest	Alternatywny plugin do obsługi Bluetootha. Pracuje tylko i wyłącznie z Bluetooth Low Energy
com.megster.cordova.bluetoothserial	jest	Jeden z bardziej popularnych pluginów, stworzonych w celu umożliwienia obsługi klasycznego Bluetooth (w przypadku Androida) i Bluetooth Low Energy (w przypadku iOS). Zaprojektowany w celu komunikacji urządzenia mobilnego z Arduino.
com.megster.cordova.rfduino	jest	Funkcja do wykrywania i łączenia się z RFduino przez Bluetooth
com.phonegap.plugins.barcodescanner	jest	Bardzo przydatny plugin umożliwiający skanowanie (a nawet tworzenie) kodów kreskowych i dwuwymiarowych. Rodzaje obsługiwanych kodów zależą od systemu operacyjnego
com.phonegap.plugins.facebookconnect	jest	Oficjalny plugin Facebooka ułatwiający logowanie użytkowników w oparciu o ich konta Facebookowe
com.phonegap.plugins.pushplugin	jest	Obsługa odbierania przychodzących powiadomień typu push
com.pushwoosh.plugins.pushwoosh	jest	Alternatywny plugin do obsługi powiadomień push, w tym również do wysyłania ich, a nie tylko odbierania
com.randdusing.bluetoothle	jest	Alternatywny plugin do obsługi Bluetootha. Pracuje tylko i wyłącznie z Bluetooth Low Energy i ma inny zestaw funkcji i możliwości, niż com.megster.cordova.bluetoothserial
com.rjfun.cordova.httpd	jest	Plugin umożliwiający uruchomienie webserwera na urządzeniu z systemem mobilnym Android lub iOS. Przydatny przy realizowaniu komunikacji przez sieć, w której to smartfon odpowiada na żądania innego urządzenia oraz do realizacji komunikacji pomiędzy różnymi aplikacjami działającymi na urządzeniu mobilnym
com.tomvanenckevort.cordova.bluetoothserial	jest	Zmodyfikowany plugin com.megster.cordova.bluetoothserial, tak by obsługiwał też możliwość inicjacji komunikacji Bluetooth ze starony urządzenia sterowanego, a nie tylko z telefonu. Niektóre funkcje pluginu wzorcowego mogą nie być obsługiwane. Uwaga! Wtyczka nie jest dostępna w oficjalnym repozytorium Cordovy
cordovarduino	brak	Wtyczka tylko dla Androida. Umożliwia prowadzenie komunikacji z użyciem interfejsu szeregowego poprzez USB. Wtyczka ta nie znajduje się w bazach projektu Cordova. Dostępna jest z adresu: https://github.com/xseignard/cordovarduino.git
de.appplant.cordova.plugin.local-notification	jest	Obsługa powiadomień systemowych, czy to w obszarze powiadomień, wyświetlanie okienek dialogowych, prezentacja dodatkowych etykietek na głównej ikonce aplikacji czy choćby odtwarzanie dźwięku. Działa również, gdy aplikacja jest uruchomiona w tle
org.chromium.storage	brak	Dostęp do pamięci urządzenia, synchronizowanej automatycznie za pomocą usługi Google Chrome sync
plugin.google.maps	brak	Bardzo zaawansowana biblioteka umożliwiająca obsługę map google w aplikacji. Tylko dla Androida i iOS
plugin.http.request	brak	Zestaw funkcji ułatwiających wysyłanie żądań HTTP
tr.bel.mamak.sms_plagin	jest	Jeden z pluginów do obsługi SMSów

smartfonu. W tym celu twórcy Cordovy wprowadzili mechanizm pluginów – wtyczek, które pozwalają na łatwe rozszerzenie zestawu funkcji platformy bez zbytniego rozbudowywania jej rdzenia. Dzięki wtyczkom osoby, które nie potrzebują np. obsługi odbiornika GPS czy wibracji, mogą kompilować programy pozbawione zbędnego dla nich kodu, a w razie, gdy zajdzie potrzeba rozbudowy aplikacji, dodanie nowych funkcji nie będzie problemem.

Pluginy

W momencie powstawania tego artykułu liczba oficjalnie dostępnych pluginów Cordovy przekraczała 680. Niektóre z nich zostały opracowane przez samych twórców platformy i są zaliczane do pluginów podstawowych. Zostały one wymienione w **tabeli 1** i obejmują przede wszystkim obsługę uniwersalnych funkcji sprzętowych, takich jak: nawigacja satelitarna, akcelerometr, powiadomienia, kamera itp. Oficjalne pluginy Cordovy to zarazem w większości wtyczki obsługiwane przez PhoneGap Build w wersji bezpłatnej, o czym wspominaliśmy w poprzedniej części kursu. Wtyczki stworzone przez niezależnych twórców, w dużej mierze służą uproszczeniu implementacji niektórych dodatkowych funkcji. Zawierają gotowe schematy graficzne i mechanizmy prezentowania informacji w sposób typowy dla urządzeń mobilnych lub ułatwiają dostęp do niektórych serwisów internetowych, takich jak np. Google Maps czy Facebook. Warto też zwrócić uwagę na pluginy ułatwiające tworzenie aplikacji o wyglądzie natywnym dla poszczególnych platform mobilnych, takich jak np. Blackberry, czy umożliwiające komunikację za pomocą alternatywnych interfejsów. W **tabeli 2** zebraliśmy ciekawe pluginy, które mogą przydać się elektronikom. Kompletny zestaw wtyczek Cordovy można znaleźć pod adresem *http://plugins.cordova.io/*, a pluginy wspierane przez bezpłatną wersję usługi PhoneGap

Listing 1. Domyślna zawartość pliku index.html

```
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="format-detection" content="telephone=no" />
        <meta name="msapplication-tap-highlight" content="no" />
        <!-- WARNING: for iOS 7, remove the width=device-width and height=device-height attributes. See
https://issues.apache.org/jira/browse/CB-4323 -->
        <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1, minimum-scale=1,
width=device-width, height=device-height, target-densitydpi=device-dpi" />
        <link rel="stylesheet" type="text/css" href="css/index.css" />
        <title>Hello World</title>
    </head>
    <body>
        <DIV style="width:80%;height:100%;margin:auto;padding-top:120px">
                <H2 style="text-align: center;max-width:100%;margin:auto;font-size: xx-large;">Kurs
programowania aplikacji mobilnych dla elektroników</H2>
                <img style="margin-left: 0%;width: 100%;" src="img/ep_logo.jpg"/>
        <div id="deviceready" class="blink" style="margin: auto;text-align: center;width: 70%; margin-
top:50px;">
            <p class="event listening">Connecting to Device</p>
            <p class="event received">Device is Ready</p>
        </div>
        </div>
        <script type="text/javascript" src="cordova.js"></script>
        <script type="text/javascript" src="js/index.js"></script>
    </body>
</html>
```

Build są wymienione na stronie pod adresem *https://build.phonegap.com/plugins*. Uwaga – zdarzają się wtyczki, które dostępne są ze stron PhoneGapa, a nie ma ich w repozytorium Cordovy. Można też samodzielnie tworzyć własne wtyczki, a nawet udostępniać je w ramach repozytoriów Cordovy.

Zdecydowana większość wtyczek ma dosyć rozbudowane opisy działania i sposobu korzystania z funkcji w aplikacjach. Pluginy przygotowane przez twórców Cordovy są szczególnie dokładnie opisane i używanie z nich nie powinno sprawiać problemu. My natomiast pokażemy najpierw, jak takie wtyczki instalować.

Instalowanie pluginów

Począwszy od Cordovy w wersji 3.0, standardowa instalacja, którą przeprowadziliśmy, nie zawiera żadnych wtyczek i trzeba je dodać samodzielnie. W tym celu z linii poleceń, z katalogu naszej aplikacji (dla przypomnienia, w naszym wypadku jest to **c:\kursEP\hello**) wydajemy polecenie:

cordova plugin add <nazwa pluginu>

Spowoduje to automatyczne pobranie najnowszej wersji wybranego pluginu i jego zainstalowanie na potrzeby naszej aplikacji, dla zainstalowanych wszystkich platform docelowych. Katalog z wtyczką pojawi się w podkatalogu **plugins** naszej aplikacji, a do pliku konfiguracji zainstalowanej platformy, również w podkatalogu **plugins**, dopisana zostanie linia deklarująca użycie zainstalowanej wtyczki. My wybraliśmy plugin **org.apache.cordova.device**, który w momencie tworzenia kursu był oferowany w wersji 0.2.13. Gdybyśmy chcieli wymusić pobranie akurat tej wersji wtyczki, musielibyśmy napisać np.:

cordova plugin add org.apache.cordova.device@0.2.13

Z poziomu linii poleceń możemy też wyszukiwać pluginy dostępne w repozytorium Cordovy, używając polecenia:

cordova plugin search <fragment nazwy pluginu>

przeglądać zainstalowane pluginy:

cordova plugin list

czy też usuwać je z danej aplikacji:

cordova plugin remove <nazwa pluginu>

W przypadku wtyczek niedostępnych w oficjalnym repozytorium Cordovy, możemy pobrać je za pomocą tego samego polecenia, ze znanego nam repozytorium Git

cordova add <url repozytorium>

lub z katalogu z dysku

cordova add <nazwa katalogu z pluginem>

Bardziej zaawansowane opcje instalacji pluginów, takie jak ich konfigurowanie pod kątem wybranych platform, można ustawić za pomocą polecenia **plugman**, którego nie będziemy teraz opisywać.

Oglądamy kod

Czas na własnoręczne napisanie pierwszego kodu. W tym celu zobaczmy, jak wygląda automatycznie wygenerowany kod naszej aplikacji, którą stworzyliśmy w poprzedniej części kursu. Plik **index.html** został pokazany na **listingu 1**. Jedyne, co w nim zmieniliśmy, to treść nagłówka H2 oraz pojawiające się logo. Nowy plik z logo wrzuciliśmy do podkatalogu **img** katalogu **www** naszej aplikacji. Warto zaznaczyć, że w znacznikach <!--- --> umieszcza się komentarze. Pod koniec sekcji HEAD znajduje się odniesienie do pliku **index.css** z podkatalogu **css**, z którego ładowane są style opisujące wygląd elementów w aplikacji. Na razie nie będziemy się nimi zajmować. Zwróćmy natomiast uwagę na linie:

```
<script type="text/javascript"
src="cordova.js"></script>
<script type="text/javascript" src="js/in-
dex.js"></script>
```

Polecenia w nich zawarte powodują załadowanie skryptów JavaScript z plików. Pierwszy z nich nie istnieje w naszej podstawowej strukturze katalogowej i jest dodawany dopiero w momencie budowania aplikacji na potrzeby konkretnej platformy docelowej. Zawiera polecenia systemowe Cordovy i nie będziemy go modyfikować. Drugi natomiast zawiera interesujące nas funkcje, właściwe dla tej, konkretnej aplikacji. To, że oba pliki zostały tak po prostu zaczytane deklaracją <SCRIPT> sprawia, że zostaną one automatycznie uruchomione wraz ze startem aplikacji. Kod pliku **index.js** został pokazany na **listingu 2**.

JavaScript jest językiem bardzo podobnym w notacji do języka C; jest prostszy i obsługuje obiekty. Linie komentarzy są poprzedzane albo podwójnymi ukośnikami, albo umieszczane w ukośnikach z gwiazdkami. Kod znajdujący się w pliku jest wykonywany automatycznie w momencie jego załadowania. W przypadku pliku **index.js** wykonywana jest najpierw deklaracja zmiennej

Listing 2. Początkowa zawartość pliku index.js

```
var app = {
    // Application Constructor
    initialize: function() {
        this.bindEvents();
    },
    // Bind Event Listeners
    //
    // Bind any events that are required on startup. Common events are:
    // 'load', 'deviceready', 'offline', and 'online'.
    bindEvents: function() {
        document.addEventListener('deviceready', this.onDeviceReady, false);
    },
    // deviceready Event Handler
    //
    // The scope of 'this' is the event. In order to call the 'receivedEvent'
    // function, we must explicitly call 'app.receivedEvent(...);'
    onDeviceReady: function() {
        app.receivedEvent('deviceready');
    },
    // Update DOM on a Received Event
    receivedEvent: function(id) {
        var parentElement = document.getElementById(id);
        var listeningElement = parentElement.querySelector('.listening');
        var receivedElement = parentElement.querySelector('.received');

        listeningElement.setAttribute('style', 'display:none;');
        receivedElement.setAttribute('style', 'display:block;');

        console.log('Received Event: , + id);
    }
};
app.initialize();
```

(obiektu) **app**, a następnie uruchamiane jest polecenie **initialize()**, będące funkcją obiektu **app**.

Standardowy przebieg inicjalizacji aplikacji rozpoczyna się od przypisania funkcji, do pojawiających się zdarzeń, w ramach funkcji **app.bindEvents()**. Domyślnie tworzone jest przypisanie zdarzenia **deviceready**, którego wystąpienie oznacza, że aplikacja jest uruchomiona i urządzenie jest gotowe do dalszej pracy. Przypisanie tworzone jest z użyciem funkcji **document.addEvent Listener()**, tak by wywołać funkcję **app.onDeviceReady()**. Ta natomiast wywołuje funkcję **app.receivedEvent()**, która tworzy zmienne, zawierające odnośniki do znajdujących się w pliku **index.html** obiektów CSS-owych klas **listening**, **reveiced** i do warstwy o identyfikatorze **deviceready**. Następnie, w tej samej funkcji, domyślnie widoczny napis „Connecting to Device" jest ukrywany, a w jego miejscu pojawia się dotąd niewidoczny (dzięki stylom CSS z pliku **index.css**) napis „Device is Ready". W końcu do konsoli systemowej przekazywana jest informacja o zaistniałym zdarzeniu.

Programowanie zdarzeniowe

Tak zawiła inicjalizacja nie jest konieczna, ale wynika z zalecanego stylu programowania i nie będziemy jej zmieniać, a jedynie rozbudowywać. Warto natomiast zauważyć, że programowanie w JavaScripcie, a więc cały kod wykonywalny, który będziemy tworzyć, opiera się o zdarzenia. Co ważne, wiele z funkcji będzie pracowało asynchronicznie, co nie gwarantuje, że wykonają się w sekwencji po sobie, tak jak są napisane. Wszystko zależy od składni i sposobu ich wywoływania. W przypadku przypisań zmiennych i prostych operacji matematycznych nie trzeba mieć obaw, że kolejne linie kodu wykonają się w innej kolejności. Sytuacja wygląda jednak inaczej, gdy w jednej linijce np. sięgamy po dane z jakiejś bazy, a w drugiej chcemy z nich skorzystać. Pobieranie danych może zakończyć się później, niż rozpocznie się wykonywanie drugiej linii kodu, co ma pewne zalety, ale utrudnia programowanie. Zaletą jest fakt, że wywoływanie takich funkcji nie blokuje interfejsu ani pracy programu. W naszym przypadku ładująca się aplikacja wykonuje wszystkie potrzebne operacje w czasie, gdy my na wyświetlaczu widzimy już podstawowy ekran programu. Gdy ładowanie się skończy, występuje zdarzenie **deviceready**, które obsługujemy przypisaną do niego funkcją. W podobny sposób będziemy przypisywać funkcje do wykonania w momencie zakończenia działania (z sukcesem lub z błędem – w zależności od potrzeb) innych, uruchamianych przez nas operacji. Niestety, czasem można zapomnieć o tej specyfice JavaScriptu, co może prowadzić do trudnych do odnalezienia błędów.

Należy też pamiętać, że w przypadku polecenia **cordova build**, kompilator wcale nie sprawdza właściwego, javascriptowego kodu aplikacji. Jest on umieszczany w gotowym pliku APK, bez jakiejkolwiek walidacji i dopiero przy włączeniu aplikacji może okazać się, że w kodzie są jakieś literówki. Dlatego dla przyspieszenia programowania, warto uruchomić sobie przeglądarkę internetową na komputerze i przed budową pliku APK otwierać w niej plik **index.html** tworzonej aplikacji. Zaglądając do okna konsoli JavaScriptu w przeglądarce można zobaczyć, czy interpreter kodu nie zwrócił żadnego błędu, a więc czy składnia jest formalnie poprawna.

Komunikacja internetowa i ethernetowa

Programowanie zaczniemy od komunikacji przez protokół TCP/IP. Korzyścią z faktu, że aplikacja jest uruchamiana na urządzeniu z pełnym systemem mobilnym jest to, że nie musimy się martwić o konfigurację standardowych interfejsów sieciowych, ani o wybór połączenia. Z punktu widzenia programisty, do naszej dyspozycji (nawet bez instalacji jakichkolwiek pluginów), dostępne jest wszystko to, co oferuje działająca przeglądarka internetowa. Oznacza to, że system samodzielnie przekieruje nasze żądania do odpowiednich interfejsów, w zależności od numerów IP, do których będziemy się odwoływać. Domyślnie w zdecydowanej większości urządzeń, jeśli dostępna będzie sieć Wi-Fi, ruch będzie kierowany przez nią, a jeśli nie – przez sieć komórkową, niezależnie od tego czy pracuje ona w trybie 2G, 3G czy 4G.

Na początek po prostu załadujmy jakąś stronę internetową, gdy tylko aplikacja się w pełni uruchomi i urządzenie będzie gotowe do pracy. W tym celu, w pliku **index.js**

tworzymy funkcję **app.openWebsite()**, która za pomocą polecenia **location.replace()** przekierowuje nas na wybraną stronę internetową. Ponadto modyfikujemy funkcję **app.onDeviceReady()**, tak by uzyskać kod jak na **listingu 3**. Uruchomienie tej aplikacji (po uprzednim skompilowaniu jej z użyciem polecenia **cordova build**) powoduje wyświetlenie strony internetowej Elektroniki Praktycznej na ekranie telefonu lub symulatora (**rysunek 1**). To teoretycznie nic specjalnego, gdyż tę samą operację moglibyśmy wykonać za pomocą przeglądarki internetowej, ale warto zauważyć, że w oknie nie mamy paska adresu, ani nie możemy korzystać z żadnych gestów (choć to zależy od ustawień). Możemy się jedynie poruszać po stronie i korzystać ze znajdujących się na niej linków, które nadal będą otwierały się w oknie aplikacji. Przypomina to działanie przeglądarki internetowej w trybie kiosk, który stosowany jest czasem tam, gdzie trzeba ograniczyć użytkownikom dostęp do nieautoryzowanych stron.

Załóżmy teraz, że za pomocą telefonu komórkowego chcemy sterować ruchem automatycznej bramy. Przyjmijmy, że jej sterownik jest podłączony do sieci lokalnej i ma prosty interfejs webowy z dwoma przyciskami. Możemy go wywołać za pomocą przeglądarki internetowej, albo stworzyć aplikację, która zaraz po uruchomieniu załaduje treść tej strony. W tym celu podmieniamy adres strony EP z listingu 3 na adres interfejsu webowego sterownika napędu bramy, uzyskując linię np.:

```
location.replace('http://192.168.0.6/brama.php');
```

Po uruchomieniu takiej aplikacji, na ekranie pojawia się interfejs webowy, którego fragment przedstawiono na **rysunku 2**. Trudno oczekiwać, że interfejs www będzie zoptymalizowany pod kątem aplikacji mobilnej, dlatego skorzystamy z innej funkcji naszego sterownika bramy. Przyjmijmy, że pozwala on na otwieranie i zamykanie bramy poprzez wysłanie odpowiedniej wartości parametru „akcja" na adres interfejsu WWW sterownika. Dostępne wartości to: „otworz" i „zamknij" oraz „stan" z czego ta trzecia pozwala dowiedzieć się, czy brama jest aktualnie otwarta czy zamknięta.

AJAX

Parametry wywołań http można przekazywać tzw. metodą GET, która polega na tym, że na końcu adresu serwera, po znaku zapytania, dokleja się listę nazw parametrów z ich wartościami (ze znakiem równości pomiędzy każdym parametrem a wartością), oddzielnych od siebie pojedynczymi znakami &. W naszym przypadku, aby otworzyć bramę, musielibyśmy wywołać adres

http://192.168.0.6/brama.php&akcja=otworz

Nie będziemy jednak w tym celu używać polecenia **location.replace()**, gdyż w żaden sposób nie poprawiłoby to naszej sytuacji, nawet jeśli serwer, po otrzymaniu parametru akcji, wyświetlałby nieco inny interfejs użytkownika. Zamiast tego skorzystamy z wywołania asynchronicznego AJAX (Asynchronous JavaScript and XML). Na całe wywołanie AJAX składają się trzy techniki: obiekt XHR (XMLHttpRequest), język JavaScript i związana z formatem otrzymywanych danych, którym – mimo nazwy, wcale nie będzie XML. Obiekt XHR jest już zaimplementowany w przeglądarkach internetowych, a więc w praktyce i w naszej aplikacji. Moglibyśmy więc po prostu wywołać polecenia:

```
var xml = new XMLHttpRequest();
xml.open(„GET", „http://192.168.0.6/brama.php&akcja=otworz", true);
xml.send();
```

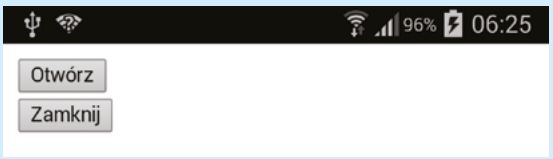
i w ten sposób spowodować otwarcie bramy. Ponieważ jednak będziemy potrzebowali wykonywać bardziej zaawansowane operacje, skorzystamy z bardzo pomocnej i popularnej, choć niemałej biblioteki jQuery. Zawiera ona szereg funkcji znacząco ułatwiających manipulacje na obiektach na stronach internetowych, co przyda nam się w trakcie budowy aplikacji.

jQuery można pobrać ze strony *http://jquery.com/download/*. W naszym przypadku pobraliśmy plik **jquery-2.1.3.min.js**, który jest zminimalizowaną wersją

Listing 3. Zmodyfikowane linie pliku index.js, tak by aplikacja ładowała stronę http://ep.com.pl

```
onDeviceReady: function() {
    app.openWebsite();
},
openWebsite: function() {
    location.replace('http://ep.com.pl');
},
```

Rysunek 1. Uruchomiona aplikacja, ładująca stronę http://ep.com.pl



Rysunek 2. Fragment ekranu uruchomionej aplikacji, ładującej interfejs www sterownika bramy

Listing 4. Fragment pliku index.html, obejmujący sekcję BODY, po dodaniu dużych przycisków sterowania bramą

```
<body>
<DIV style="width:80%;height:100%;margin:auto;padding-top:120px">
    <div id="deviceready" class="blink" style="margin: auto;text-align: center;width: 70%; margin-top:50px;">
        <p class="event listening">Connecting to Device</p>
        <p class="event received">Device is Ready</p>
    </div>
</div>
<script type="text/javascript" src="cordova.js"></script>
<script type="text/javascript" src="js/jquery-2.1.3.min.js"></script>
<script type="text/javascript" src="js/index.js"></script>
<div style="display:table;width:100%;height:100px;background-color:green;font-size:xx-large;text-
align:center">
    <div id="openButton" style="display:table-cell; vertical-align: middle;">OTWÓRZ</div>
</div>
<div id="closeButton" style="display:table;width:100%;height:100px;background-color:red;font-size:xx-
large;text-align:center">
    <div style="display:table-cell; vertical-align: middle;">ZAMKNIJ</div>
</div>
</body>
```

jQuery w wersji 2.1.3. Plik umieszczamy w podkatalogu **www\js** aplikacji i dopisujemy linijkę:

```
<script type="text/javascript" src="js/jquery-2.1.3.min.js"></script>
```

w pliku **index.hml**, pomiędzy linijkami wczytującymi biblioteki Cordovy i plik **index.js**.

Możemy teraz skorzystać z polecenia **jQuery.get()**, które jest uproszczoną formą wywołania AJAX z parametrami przekazywanymi metodą GET. Ponieważ w praktyce biblioteka jQuery bywa używana w aplikacjach webowych bardzo obficie, przyjęło się, że standardowo definiowany jest skrót w postaci znaku **$**, który zastępuje słowo **jQuery** w wywołaniach funkcji. Oznacza to, że będziemy używać zapisu **$.get()**.

Do kontrolowania bramy stworzymy w naszej aplikacji dwa duże przyciski, które będą powodowały wysłanie odpowiednich żądań AJAXowych do sterownika. Z pliku **index.html** usuwamy niepotrzebne elementy graficzne i dodajemy linijki z nowymi warstwami, które posłużą nam za zielony i czerwony przycisk sterujący. Sekcja **BODY** pliku **index.html** wygląda wtedy tak, jak na **listingu 4**. W pliku **index.js** dodajemy nowe funkcje: **app.assignButtons()** i **app.sendGateCommand()** oraz modyfikujemy funkcję **app.onDeviceReady()**, tak by wywoływała polecenie przypisania działań do przycisków, tak jak to zostało przedstawione na **listingu 5**. W efekcie, uruchomiona aplikacja wygląda tak, jak na **rysunku 3**.

Warto przeanalizować funkcje **assignButtons()** i **sendGateCommand()**. Pierwsza z nich czeka na załadowanie się całego kodu strony z pliku **index.html** i innych, których pobranie wynika z kodu **index.html**. W tym celu zastosowano polecenie

$(document).ready()

Gdyby nie czekać, mogłaby zaistnieć sytuacja, w której przypisanie funkcji do przycisków zostanie wykonane, zanim jeszcze przyciski zostaną załadowane, w efekcie czego przypisanie po prostu by się nie udało. Polecenia, wykonywane po załadowaniu się kompletnej strony umieszczone są w nawiasach funkcji **ready()**. W praktyce umieszcza się tam deklarację funkcji do wywołania, wraz z jej treścią. W naszym przypadku umieszczamy dwa polecenia. Pierwsze znajduje w treści strony aplikacji obiekt o identyfikatorze **openButton** i przypisuje do zdarzenia kliknięcia w ten obiekt funkcję wywołującą funkcję **sendGateCommand()** z parametrem „otworz". Drugie wykonuje analogiczną operację dla warstwy o identyfikatorze **closeButton**, czyli dla naszego dużego, czerwonego przycisku.

Polecenie **sendGateCommand()** będzie więc wywoływane zawsze, gdy naciśniemy zielony lub czerwony przycisk, tyle że z różnymi parametrami. W definicji funkcji **sendGateCommand()** znalazła się deklaracja zmiennej z adresem, pod który ma być wysyłane żądanie AJAX oraz samo wywołanie AJAXowe pod podany adres, z parametrem przekazywanym jako argument funkcji **sendGateCommand()** i z deklaracjami funkcji, które mają się wydarzyć w przypadku sukcesu lub niepowodzenia komunikacji ze sterownikiem bramy. Warto zwrócić uwagę na sposób wywołania polecenia **$.get()**. Jako pierwszy argument podawany jest adres wywołania,

Listing 5. Funkcje do obsługi przycisków otwierania i zamykania bramy, znajdujących się w kodzie na listingu 4. Warto zauważyć, że w języku JavaScript, łańcuchy znaków łączy się ze sobą za pomocą symbolu dodawania

```
onDeviceReady: function() {
    app.receivedEvent('deviceready');
    app.assignButtons();
},

assignButtons: function() {
    $(document).ready(function() {
            $('#openButton').click(function() {
            app.sendGateCommand(„otworz");
        });
        $('#closeButton').click(function() {
            app.sendGateCommand(„zamknij");
        });
    });
},

sendGateCommand: function(command) {
    var adres="http://192.168.0.6/brama.php";
    $.get( adres, { akcja: command } )
        .done(function( data ) {
            alert( „Polecenie „ + command + „ przesłane pomyślnie" );
        })
        .error(function( data ) {
            alert( „Nie udało się wysłać polecenia „ + command);
        });
},
```

Rysunek 3. Aplikacja z dużymi przyciskami do sterowania bramą

a jako drugi, lista przekazywanych, oddzielonych przecinkami parametrów, podana w nawiasach klamrowych, złożona z nazw parametru i ich wartości, oddzielonych od siebie dwukropkiem. Co ciekawe, po funkcji **$.get()** nie pojawia się średnik, ale kropka i funkcja **$.done()**, a następnie **$.error()**. Wynika to ze specyfiki biblioteki jQuery, która pozwala na wygodne kolejkowanie odnoszących się do siebie funkcji. W tym przypadku, taka deklaracja pozwala określić, co ma się zdarzyć w przypadku pomyślnego ukończenia wywołania AJAX GET, a co jeśli wywołanie się nie uda. My w obu sytuacjach wywołujemy systemowe okienko dialogowe z adekwatnym komunikatem, korzystając ze standardowej funkcji **alert()**.

Ograniczenia domenowe

W tym momencie wypada się zastanowić, dlaczego wywołanie AJAX GET mogłoby się nie udać. Oczywistymi przyczynami będzie błąd w komunikacji, niedostępność sieci lub brak odpowiedzi ze strony serwera. To jednak nie wszystko. Standardowo w wielu przeglądarkach internetowych implementowane jest zabezpieczenie, uniemożliwiające wywoływanie zapytań JavaScriptowych, kierowanych pod adresy w innej domenie, niż strona internetowa, na której znajdują się uruchomione skrypty. To, czy w naszej aplikacji żądanie zostanie zablokowane, będzie zależało od systemu mobilnego, na którym uruchamiamy aplikację. Na wszelki wypadek warto więc postarać się ominąć to zabezpieczenie, co można zrobić na kilka sposobów. Można np. zadbać o to, by aplikacja miała dostęp do domeny, do której się odwołuje (by była zadeklarowana, jako wywodząca się z niej), można skorzystać z techniki CORS (Cross-Origin Resource Sharing), ale ta wymaga by obsługiwał ją serwer, do którego się odwołujemy oraz można zastosować sztuczkę opartą na technice JSONP. My za chwilę zastosujemy tę ostatnią, ale najpierw rozbudujemy naszą aplikację o obsługę wspomnianej wcześniej funkcji „stan" sterownika bramy.

JSON

Dodajemy w pliku index.html kolejną warstwę, która posłuży jako przycisk do sprawdzania stanu bramy:

```
<div id="stateButton" style="display:t
able;width:100%;height:100px;backgrou
nd-color:white;font-size:xx-large;text-
align:center">
<div style="display:table-cell; vertical-
align: middle;">STAN</div>
</div>
```

Rozbudowujemy funkcję **app.assignButtons()** o przypisanie odpowiedniej akcji do nowej warstwy, dodają linijki:

```
$('#stateButton').click(function() {
        app.sendGateCommand(„stan");
});
```

Natomiast w funkcji **app.sendGateCommand()**, w sekcji **done()**, skorzystamy z wcześniej nieużywanego parametru **data** funkcji wywoływanej po pomyślnym ukończeniu żądania **$.get()**. Zawiera on całą treść odpowiedzi, uzyskaną z serwera. Gdybyśmy poleceniem **$.get()** wywołali po prostu adres *http://192.168.0.6/brama.php* bez parametrów, parametr **data**, dostępny w sekcji **done()** zawierałby kod HTML z małymi przyciskami, wyświetlanymi w ramach interfejsu www przez sterownik bramy (takiego jak na rysunku 2). Nasz sterownik zachowuje się jednak zupełnie inaczej, gdy na adres *http://192.168.0.6/brama.php* przekaże mu się parametr „akcja", a mianowicie, gdy wartością tego parametru będzie wyraz „stan", wtedy w odpowiedzi otrzymamy informację o aktualnym stanie bramy.

Co więcej, nasz sterownik zwraca informację o stanie w formacie JSON, co się dobrze składa, bo to bardzo wygodny format do przetwarzania danych za pomocą języka JavaScript. JSON pozwala przekazywać obiekty zawierające zmienne w formacie:

```
{„nazwa pierwszej zmiennej": "wartość
pierwszej zmiennej", „nazwa drugiej
zmiennej": "wartość drugiej zmiennej"}
```

Liczba tak przekazywanych zmiennych jest nieograniczona, a ponadto zmiennymi mogą być kolejne obiekty JSON, co prowadzi do powstawania zagnieżdżonej struktury z wieloma nawiasami. Tworzenie i dekodowanie obiektów w formacie JSON jest łatwe, zarówno z poziomu JavaScriptu, jak i innych języków, takich jak np. PHP, Java, C, C++, Python czy Ruby. Ponadto, korzystając z biblioteki jQuery, możemy użyć polecenia **$.getJSON()** zmiast **$.get()**, dzięki czemu parametr data, który będzie dostępny w sekcji **done()** będzie już przetworzonym obiektem JavaScriptowym, a nie tylko ciągiem znaków w formacie JSON. Warto dodać, że funkcja **$.getJSON()** dodatkowo sprawdza, czy obiekt JSON jest poprawnie skonstruowany i jeśli nie jest – kończy się

Listing 6. Zmodyfikowana funkcja sendGateCommand, umożliwiająca odczyt stanu bramy funkcją $.getJSON()

```
sendGateCommand: function(command) {
    var adres="http://192.168.0.6/brama.php";
    $.getJSON( adres, { akcja: command } )
        .done(function( data ) {
            var response = „Polecenie „ + command + „ przesłane pomyślnie”
            if (‘stan’ in data) response+=”. Stan bramy to: „+data.stan;
            alert(response);
        })
        .error(function( data ) {
            alert( „Nie udało się wysłać polecenia „ + command);
        });
},
```

niepowodzeniem, skutkując wyzwoleniem akcji z sekcji **error()**, a nie **done()**.

Nasz sterownik bramy działa poprawnie i w przypadku, gdy brama jest otwarta, na żądanie akcji „stan” odpowiada generując ciąg znaków:

```
{„stan”: „brama otwarta”}
```

My natomiast modyfikujemy kod sekcji **done()** funkcji **app.sendGateCommand()**, tak by w przypadku otrzymania informacji o stanie bramy, wyświetlać ją wraz z komunikatem o powodzeniu wysłania polecenia do sterownika. Do wartości zmiennej **stan** obiektu **data** w sekcji **done()** odwołujemy się pisząc:

```
data.stan
```

Zmodyfikowana funkcja **app.sendGateCommand()** została przedstawiona na **listingu 6**, a rezultat działania programu na **rysunku 4**.

JSONP

Powyżej opisane wywołanie AJAX GET JSON również podlega zabezpieczeniu, o którym poinformowaliśmy wcześniej. Ale znajomość formatu JSON pozwala, po drobnym jego rozbudowaniu, ominąć ten problem. Okazuje się bowiem, że zabezpieczenie ograniczające dynamiczne wywołania do obcych domen obejmuje jeden ważny wyjątek. System praktycznie zawsze pozwala na ładowanie skryptów JavaScriptowych z dowolnych serwerów. Gdyby więc nasz sterownik bramy zwracał informacje o stanie w postaci skryptu, a nie danych w formacie JSON, moglibyśmy przekazywać mu żądania, niezależnie od tego w jakiej domenie się znajduje. Problem w tym, by treść ładowanego w ten sposób skryptu odnosiła się jakoś do aktualnie wywoływanego kodu. W tym celu wymyślono technikę komunikacji JSONP, którą tłumaczy się jako „JSON with Padding”. W skrócie, polega ona na tym, że wśród parametrów przekazywanych serwerowi, podajemy dodatkowo nazwę funkcji, jaka ma być wywołana przez skrypt generowany przez serwer w odpowiedzi na nasze żądanie. Natomiast właściwa treść odpowiedzi jest zwracana jako parametr tej funkcji, wywoływanej w otrzymanym od serwera skrypcie.

Czyli zamiast wywołania:

http://192.168.0.6/brama.php&akcja=stan

generujemy wywołanie:

http://192.168.0.6/brama.php&akcja=stan&jsoncallback=podajstan

a w odpowiedzi zamiast

{„stan”: „brama otwarta”}

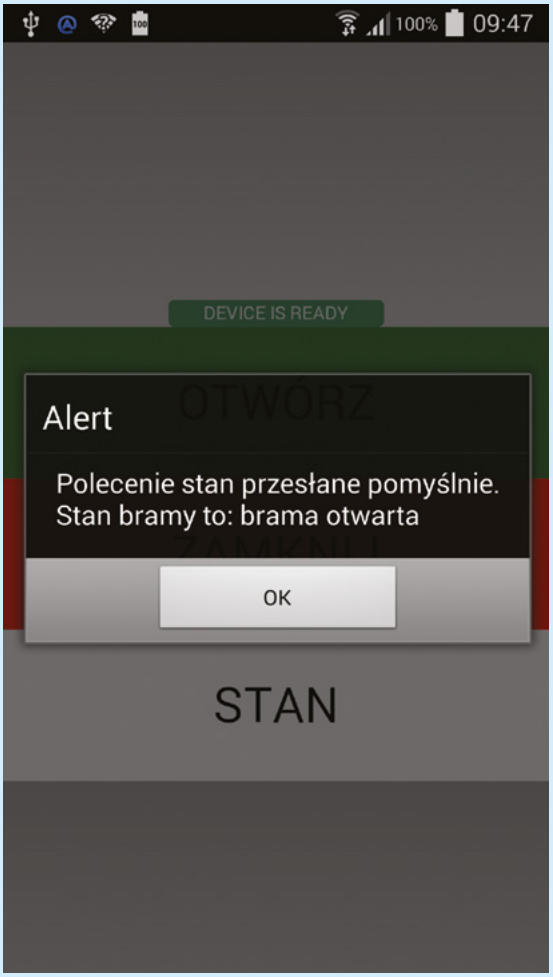
otrzymujemy

podajstan({„stan”: „brama otwarta”});

Musimy się tylko upewnić, że funkcja podajstan() jest dostępna dla wszystkich wczytywanych skryptów. Jak to wszystko zrobić? Znów skorzystamy z dobrodziejstwa jQuery, które bardzo automatyzuje wywołania JSONP. Wystarczy, że zmodyfikujemy wartość zmiennej adres:

```
var adres="http://192.168.0.6/brama.php?jsoncallback=?";
```

co sprawi, że funkcja **$.getJSON()** automatycznie potraktuje to żądanie jako komunikację JSONP. Biblioteka jQuery samodzielnie stworzy nową, publiczną funkcję o losowej nazwie, której wywołanie z dowolnego skryptu będzie prowadziło do uruchomienia kodu z sekcji **.done()** funkcji **$.getJSON()**. Nazwa tej automatycznie powstałej funkcji zostanie przekazana do serwera jako parametr o nazwie „jsoncallback” i musi być przez serwer odpowiednio doklejona do zwracanych danych JSON, tworząc odpowiedź JSONP. Gdyby nazwa funkcji potrzebnej do umieszczenia w formacie JSONP nie mogła być przekazywana akurat w ramach parametru o nazwie jsoncallback, albo gdyby musiała być z góry zdefiniowana, należy skorzystać z bardziej zaawansowanej funkcji **$.ajax()**, która pozwala dowolnie definiować parametry wywołań AJAXowych.



Rysunek 4. Odczytywanie stanu bramy ze sterownika

POST

Omawiając komunikację internetową warto wspomnieć o bezpieczeństwie. Załóżmy, że do otwarcia bramy, konieczne jest podanie kodu, który – dla uproszczenia – został na stałe zaszyty w naszej aplikacji. Załóżmy też, że brama nie znajduje się w naszej sieci lokalnej, tylko sterujemy nią poprzez publiczną sieć internetową. Korzystając z żądań AJAX GET, wszystkie przekazywane serwerowi parametry są jawnie podawane w adresie serwera. Adresy te są nierzadko zapisywane przez różnego rodzaju routery i bramki sieciowe i gdybyśmy chcieli uniknąć sytuacji, że kod do bramy zostanie zapisany w rejestrach bramek, przez które płynie ruch, musimy skorzystać z żądań AJAX POST. Ponadto niektóre serwery z góry obsługują tylko jeden rodzaj przekazywanych parametrów (GET lub POST), a więc dobrze być przygotowanym na konieczność posłużenia się tą drugą metodą.

Biblioteka jQuery znów przychodzi nam z pomocą. Teoretycznie wystarczyłoby zastąpić polecenie **$.get()** poleceniem **$.post()**, ale że nasz serwer odpowiada danymi w formacie JSON (załóżmy, że tym razem nie JSONP), nie wystarczy zastąpić funkcji **$.getJSON()**, funkcją **$.postJSON()**, gdyż taka nie istnieje. Dlatego zastępujemy funkcję **$.getJSON()** funkcją **$.post()** oraz zmieniamy jej parametry wywołania – wprowadzając dodatkowy kod do przekazania serwerowi, dodając **null** jako trzeci parametr i **„json”** jako czwarty:

```
$.post( adres, { akcja: command, kod:
„12345” }, null, ”json”)
```

Alternatywnie, w miejscu **null** moglibyśmy umieścić deklarację funkcji, która wykonywana jest w przypadku sukcesu żądania, zamiast umieszczać ją w sekcji **done()**.

Przywracamy też adres, tak by nie próbował skorzystać z komunikacji JSONP:

```
var adres=”http://192
.168.0.6/brama.php”;
```

Dlaczego wracamy do typowej wymiany danych w formacie JSON, zamiast dalej korzystać z JSONP? Niestety, jest to konieczne, jeśli chcemy przekazywać parametry niejawnie metodą POST. JSONP to tylko sztuczka – wykorzystanie luki w zabezpieczeniach, które nie działają gdy ładowany jest skrypt z dowolnego adresu internetowego. Rzecz w tym, że ładując skrypty nie można przekazywać parametrów metodą POST, a więc JSONP nie będzie działać z POSTem. Oznacza to, że aby móc przesyłać kod do bramy niejawnie, konieczne jest zapewnienie zgodności domen sterownika bramy i aplikacji lub implementacja wcześniej wspomnianego mechanizmu CORS na sterowniku.

Musimy dodać, że samo użycie metody POST zamiast GET wcale nie zabezpiecza przed osobami celowo podsłuchującymi ruch sieciowy, w celu znalezienia np. haseł do różnych usług. Dopiero użycie protokołu SSL, czyli wywołań HTTPS zamiast HTTP daje względnie dużą dozę bezpieczeństwa. Jednakże protokół ten musi być obsłużony po stronie serwera, czyli w tym wypadku – sterownika bramy.

Odczytywanie informacji o urządzeniu

Teraz wykorzystamy plugin, zainstalowany na wstępie tej części kursu. Zastąpimy stały kod (hasło) przekazywany do sterownika, unikalnym identyfikatorem urządzenia mobilnego, z którego wysyłane jest żądanie otwarcia lub zamknięcia bramy. Identyfikator ten może być np. zapisywany w historii sterownika i może stanowić podstawę do udzielania lub odmowy zgody na otwarcie bramy.

Unikalne ID urządzenia pobieramy korzystając z pluginu **org.apache.cordova.device**. W momencie, gdy go zainstalowaliśmy, w naszej aplikacji pojawiła się globalna zmienna **device**, a która podczas inicjalizacji programu wypełniana jest podstawowymi informacjami o sprzęcie i systemie operacyjnym, na którym działa aplikacja. Zmienna **device** zawiera 5 atrybutów:

1. **device.cordova** – numer wersji platformy Cordova, użytej w aplikacji (np. 3.6.4),
2. **device.model** – model urządzenia (np. GT-I9505 dla Samsunga Galaxy S4),
3. **device.platform** – system operacyjny (np. Android),
4. **device.uuid** – unikalny identyfikator w postaci ciągu 16 znaków, odpowiadającego 64-bitowej liczbie zapisanej w systemie szesnastkowym,
5. **device.version** – wersja systemu operacyjnego (np. 4.4.2).

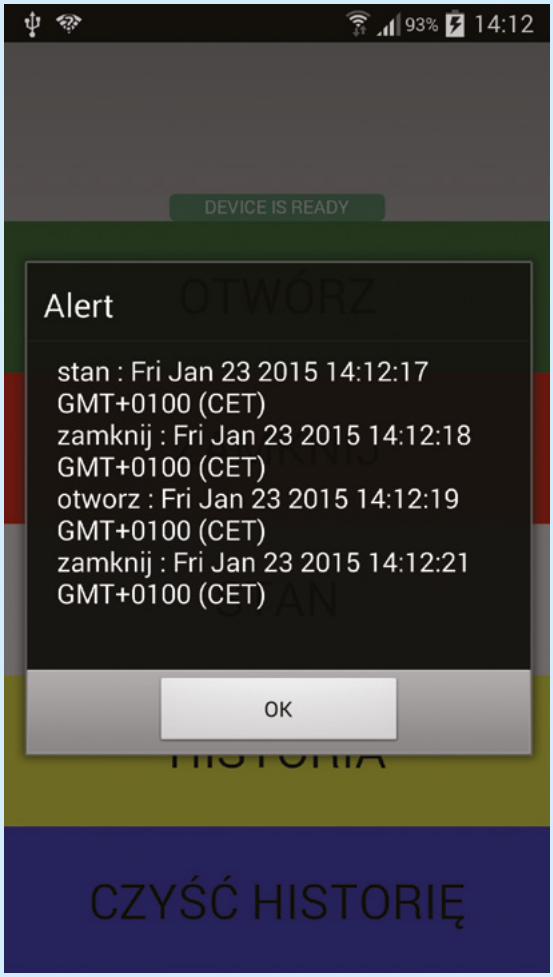
Zmienna **device** uzupełniana jest jeszcze o 6. atrybut **available**, który informuje o tym, czy dane dotyczące urządzenia zostały już wypełnione.

Listing 7. Zmodyfikowane funkcje przygotowane do obsługi historii zapisywanej w localStorage

```
assignButtons: function() {
    $(document).ready(function() {
        $('#openButton').click(function() {
            app.sendGateCommand(„otworz”);
        });
        $('#closeButton').click(function() {
            app.sendGateCommand(„zamknij”);
        });
        $('#stateButton').click(function() {
            app.sendGateCommand(„stan”);
        });
        $('#showButton').click(function() {
            app.showHistory();
        });
        $('#clearButton').click(function() {
            app.clearHistory();
        });
    });
},
sendGateCommand: function(command) {
    var adres=”http://192.168.0.6/brama.php”;
    $.post( adres, { akcja: command, kod: device.uuid }, null,'json')
        .done(function( data ) {
            var response = „Polecenie „ + command + „ przesłane pomyślnie”
            if ('stan' in data) response+=”. Stan bramy to: „+data.stan;
            alert(response);
            var d = new Date();
            var olddata=window.localStorage.getItem(„data”);
            if (olddata!=null) info=olddata+command+” : „+d.toString()+”\n”;
            else info=command+” : „+d.toString()+”\n”;
            window.localStorage.setItem(„data”,info);
        })
        .error(function( data ) {
            alert( „Nie udało się wysłać polecenia „ + command);
        });
},
showHistory: function() {
    alert(window.localStorage.getItem(„data”));
},
clearHistory: function() {
    window.localStorage.clear();
    alert(„Wyczyszczono historię”);
},
```

Listing 8. Dodatkowe przyciski w pliku index.html, przeznaczone do wyświetlania i czyszczenia historii. Wraz z ich dodaniem zmniejszono też odstęp w pierwszej warstwie sekcji BODY do 50 pikseli (padding-top:50px)

```
<div id="showButton" style="display:table;width:100%;height:100px;background-color:yellow;font-size:xx-large;text-align:center">
    <div style="display:table-cell; vertical-align: middle;">HISTORIA</div>
</div>
<div id="clearButton" style="display:table;width:100%;height:100px;background-color:blue;font-size:xx-large;text-align:center">
    <div style="display:table-cell; vertical-align: middle;">CZYŚĆ HISTORIĘ</div>
</div>
```



Rysunek 5. Pełna aplikacja z trwałym zapisem historii poleceń wydawanych do sterownika bramy

Skorzystanie z tych danych jest niezmiernie proste. W naszym przypadku wystarczy tylko podmienić ciąg „12345" na **device.uuid**, tak by uzyskać:

```
$.post( adres, { akcja: command, kod: device.uuid }, null,'json')
```

Należy jednak zaznaczyć, że o ile na Androidzie unikalne ID jest stałe dla danego urządzenia, w przypadku innych systemów może być z tym różnie. Przykładowo, na iOSie identyfikator ten zmienia się wraz z aktualizacją systemu, a może też wraz z aktualizacją naszej aplikacji.

Zapisywanie danych w telefonie

Na koniec postaramy się trwale zapisywać dowolne dane w telefonie i je odczytywać. Konkretnie, będziemy rejestrować polecenia wydawane do sterownika bramy, wraz z godzinami ich wysłania. W tym celu skorzystamy z najprostszego i chyba najbardziej uniwersalnego sposobu trwałego zapisu danych na potrzeby aplikacji, czyli mechanizmu **localStorage**. Jest on natywnie obsługiwany przez Cordovę, gdyż jest to mechanizm zaimplementowany w nowoczesnych przeglądarkach internetowych.

LocalStorage przechowuje tablicę zmiennych, na którą składają się nazwy zmiennych i ich wartości. Nazwy są więc kluczami tablicy. Dostęp do **localStorage** odbywa się przez obiekt **window.localStorage**. Ma on 5 funkcji:

1. **setItem(„klucz", „wartość")** – przypisuje „wartość" do zmiennej o nazwie „klucz",
2. **getItem(„klucz")** – pobiera aktualną wartość zmiennej „klucz",
3. **removeItem(„klucz")** – usuwa z tablicy **localStorage** zmienną o podanym kluczu,
4. **key(n)** – zwraca n-ty klucz w tablicy **localStorage**; kolejne klucze są numerowane, a numeracja rozpoczyna się od zera,
5. **clear()** – czyści całą tablicę **localStorage**.

My będziemy przechowywać historię komend sterownika bramy w zmiennej o kluczu „data", dodając do niej za każdym razem nową linijkę, zawierającą nazwę wydanego polecenia i datę wywołania. Aktualną datę pobieramy korzystając z komendy

```
var d = new Date();
```

i zamieniamy ją na łańcuch znaków używając polecenia:

```
d.toString();
```

W ten sposób będziemy tworzyć zmienną **info**, doklejając ją do wcześniej wczytanej dotychczasowej zawartości zmiennej **localStorage** o kluczu „data":

```
info=olddata+command+" : „+d.toString()+"\n";
```

którą następnie będziemy zapisywać do pamięci lokalnej poleceniem:

```
window.localStorage.setItem(„data",info);
```

Warto zaznaczyć, że pamięć ta jest dostępna tylko i wyłącznie dla danej aplikacji.

Zmodyfikowany kod funkcji **app.sendGate Command()**, nowe funkcje do prezentacji i czyszczenia historii oraz zmodyfikowaną funkcję **app.assignButtons()**, rozszerzoną na potrzeby obsługi dodatkowych przycisków, pokazaliśmy na **listingu 7**, a dodatkowe warstwy z nowymi przyciskami z pliku **index.html**, zaprezentowaliśmy na **listingu 8**. Natomiast na **rysunku 5** widać kompletną aplikację z wyświetlonym komunikatem z historią zdarzeń.

Podsumowanie

W niniejszej części kursu pokazaliśmy, jak korzystać z pluginów Cordovy, prowadzić komunikację za pomocą sieci Ethernet, przekazywać żądania do serwerów, unikając przy tym ograniczeń wynikających z zabezpieczeń oraz jak odnosić się do niektórych zasobów urządzenia mobilnego. W kolejnej części kursu pokażemy, jak użyć innych podzespołów urządzenia: odbiornika nawigacji satelitarnej, akcelerometru, wibracji itp., czyli jak korzystać z różnych pluginów.

Marcin Karbowniczek, EP