Programowanie aplikacji mobilnych (5)

Powiadamianie użytkownika

W dotychczasowych częściach kursu programowania aplikacji mobilnych pokazaliśmy, jak skorzystać z niektórych modułów sprzętowych, dostępnych w telefonie lub tablecie, jak wysyłać proste zapytania przez Internet i jak wysyłać dane przez Bluetooth. W tym odcinku chcielibyśmy pokazać, jak usprawnić interakcję z użytkownikiem dzięki różnym formom powiadomień i okien dialogowych. Co więcej, na potrzeby przykładu zaangażujemy też interfejs Bluetooth, dzięki czemu uzupełnimy dotychczasowy kod prezentujący sposób wysyłania danych przez Bluetooth o kod służący do odbierania informacji poprzez ten interfejs.

Tak jak wcześniej, skorzystamy z przykładu ze sterownikiem bramy. Przyjmijmy, że napęd bramy może być kontrolowany nie tylko z użyciem pilota w postaci smartfona, ale i w inny sposób – tak jak to zostało przedstawione na **rysunku 1**. Pilot posłuży nam natomiast w roli bezprzewodowego narzędzia do monitorowania stanu bramy. Przygotujemy aplikację, dzięki której otwarcie lub zamknięcie bramy będzie powodowało wygenerowanie odpowiedniego komunikatu na pilocie. Dane będą przesyłane poprzez interfejs Bluetooth, ale nic nie stoi na przeszkodzie, by odbierać je np. poprzez Internet. Zaczniemy od powiadamiania użytkownika za pomocą wibracji.

Wtyczka do wibracji

Dyskretna wibracja to bardzo wygodny sposób na informowanie użytkownika o zaistniałych zdarzeniach. W wielu wypadkach pozwala użytkownikowi na rozpoznanie zdarzenia, nawet bez sięgania po telefon i w sposób ukryty przed osobami postronnymi. Noszone w kieszeni urządzenie mobilne może wibrować na różne sposoby i w różnym rytmie, który można przypisać określonym zjawiskom. Co prawda, takie zaawansowane sterowanie wibracjami dostępne jest w zależności od zainstalowanego systemu operacyjnego, ale tak czy inaczej, operowanie wibracjami jest bardzo proste i wygodne.

Aby sterować wibracjami telefonu konieczna jest instalacja odpowiedniej wtyczki. Użyjemy w tym celu wtyczki **org.apache.cordova.vibration**. Instalujemy ją poleceniem:

cordova plugin add org.apache.cordova.vibration

W naszym przypadku pobrała się wtyczka w wersji 0.3.13. W wersji tej istotne jest tak naprawdę tylko jedno polecenie, które można wywołać na dwa sposoby (z dwoma rodzajami parametrów). Jest to:

navigator.vibrate(time)

lub:

navigator.vibrate(pattern)

Przy czym **time** to czas wyrażony w postaci liczby całkowitej, odpowiadającej milisekundom czasu trwania wibracji, a **pattern** to tablica liczb całkowitych, odpowiadających naprzemiennie czasom trwania wibracji i czasom przerw pomiędzy wibracjami.

Czyli przykładowo, chcąc wywołać 1-sekundowe powiadomienie wibracyjne, użyjemy polecenia:

navigator.vibrate(1000);

a chcąc uruchomić trzy, krótkie, następujące po sobie impulsy wibracji wywołamy polecenie:

navigator.vibrate([200,100,200,100,200]);

które spowoduje wywołanie trzech 200-milisekundowych impulsów z odstępami co 100 ms.

Trzeba jednak zaznaczyć, że w przypadku systemu operacyjnego iOS, liczby podane w nawiasie nie mają znaczenia, gdyż wibracja zawsze będzie trwałą z góry określony, standardowy czas, na który programista nie ma wpływu. Natomiast parametr w postaci tablicy jest obsługiwany tylko i wyłącznie w systemach Android i Windows.

Odbiór danych przez Bluetooth

W naszym przykładzie będziemy uruchamiać wibracje po uprzednim otrzymaniu danych przez Bluetooth z napędu bramy. Kodu programu napędu nie będziemy prezentować – pozostawiamy stworzenie go użytkownikom, w oparciu o wiedzę z poprzedniej części kursu. Natomiast w kodzie pilota skorzystamy z polecenia **bluetoothSerial.subscribeRawData()**, opisanego w poprzedniej części kursu.

Rysunek 1. Schemat przykładowego systemu obsługi bramy, gdzie pilot służy do monitoringu zdarzeń związanych z bramą

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

Listing 1. funkcja app.bt1(), rozbudowana o obsługę danych przesyłanych przez interfejs Bluetooth z napędu bramy i wzbudzająca wibracje, zależne od tego, czy brama została zamknięta, czy otwarta.

```
bt1 : function(){
    bluetoothSerial.discoverUnpaired(function(devices) {
        devices.forEach(function(device) {
            if (device.name==”Brama”){
                app.btaddr=device.address;
                bluetoothSerial.connect(app.btaddr,
                    function(){
                        alert(„połączono”);
                        bluetoothSerial.subscribeRawData(
                            function(data){
                            var bytes = new Uint8Array(data);
                            if (bytes[0]==1) navigator.vibrate(1000);
                            else if (bytes[0]==2)
                                navigator.vibrate([200,100,200,100,200]);
                        },
                            function(){
                                alert(„koniec”);
                            }
                        );
                    },
                    function(){
                        alert(„zakończono połączenie”);
                    }
                );
            }
        })
    }, function(){
        alert(„error”);
    });
},
```

Powyższe polecenie pozwala na cykliczne odbieranie danych przez interfejs Bluetooth i będziemy je wykorzystywali do odbierania sygnałów o otwarciu lub zamknięciu bramy. Jeśli brama zostanie otwarta, będziemy generować trzy krótkie impulsy wibracji, a jeśli będzie zamknięta – wygenerujemy pojedynczą, jednosekundową wibrację. Skorzystamy z kodu z poprzedniej części kursu, w efekcie czego na **listingu 1** pokazujemy jedynie rozbudowaną funkcję **app.bt1()**, która uruchamiana jest po starcie aplikacji i służy do inicjacji obsługi interfejsu Bluetooth. W naszym przykładzie założyliśmy, że napęd bramy przesyła liczbę „1”, gdy brama jest zamykana i „2” w przypadku otwarcia bramy.

Warto zauważyć, że tak stworzony program pozwala nie tylko odbierać dane przez Bluetooth, gdy aplikacja jest zminimalizowana, ale też generować wtedy wibracje. Sprawia to, że jeśli nasz program działa w tle, użytkownik może swobodnie korzystać ze swojego urządzenia mobilnego i wciąż otrzymywać powiadomienia wibracyjne, o ile tylko brama zostanie otwarta lub zamknięta.

Zaawansowane powiadomienia lokalne

Wibracje, choć stanowią atrakcyjny sposób powiadamiania użytkownika, są tylko jedną z wielu metod. Do obsługi innych lokalnych powiadomień, czyli takich, które inicjowane są z danego urządzenia mobilnego, a nie wymuszane zdalnie, przydatna jest zaawansowana wtyczka **de.appplant.cordova.plugin.local-notification**. Jej działanie jest mocno uzależnione od systemu operacyjnego, gdyż poszczególne z systemów obsługują różne rodzaje powiadomień. Aktualnie omawiany plugin (w wersji 0.8) działa na platformach iOS, Android, Windows 8.1 i Windows Phone 8.1, ale w razie potrzeby można zainstalować wersję 0.7, która obsługiwała platformę WP8.

Tworzenie lokalnych powiadomień opiera się o mechanizm dodawania ich do harmonogramu systemowego. Harmonogram może obejmować powiadomienia przeznaczone do błyskawicznego wyświetlenia i odłożone w czasie; może zawierać powiadomienia pojedyncze i cykliczne. Co więcej, możliwe jest jednoczesne dodawanie do harmonogramu kilku powiadomień, usuwanie ich i podmienianie.

Same powiadomienia – w najbardziej zaawansowanej formie – mogą mieć trzy postaci:

- wyświetlenie informacji w obszarze powiadomień systemu,
- odtworzenie dźwięku,
- dodanie plakietki na ikonkę aplikacji,

przy czym ta trzecia opcja, aby mogła być swobodnie wykorzystana w Androidzie, wymaga doinstalowania oddzielnej wtyczki, której w tym przykładzie nie będziemy używać.

Instalacja wtyczki powiadomień lokalnych nie odbiega niczym od instalacji innych pluginów:

cordova plugin add de.appplant.cordova.plugin.local-notification

W naszym przypadku zainstalowała się wersja 0.8.1. Uwaga – po instalacji wtyczki, w przypadku systemu Android, wśród wymaganych uprawnień aplikacji pojawi się żądanie umożliwienia automatycznego uruchamiania jej po włączeniu urządzenia.

Po pomyślnej instalacji, w aplikacji pojawia się nowy obiekt **cordova.plugins.notification.local** z licznymi dostępnymi funkcjami. Oto one:

- **schedule(powiadomienie)** – powoduje dodanie do harmonogramu nowego powiadomienia lub szeregu powiadomień, jeśli parametrem jest tablica;
- **update(powiadomienie)** – powoduje modyfikację zapisanego w harmonogramie powiadomienia, w oparciu o unikalny identyfikator powiadomienia; tu także możliwe jest podanie tablicy jako parametru, w celu jednoczesnej aktualizacji wielu powiadomień;
- **clear(id_powiadomienia, sukces)** – powoduje wyczyszczenie wyzwolonych powiadomień, tj. usunięcie ich z centrum powiadomień (czyli obszaru powiadomień), w oparciu o ich unikalne identyfikatory i wywołanie funkcji podanej jako parametr sukces;
- **clearAll(sukces)** – powoduje usunięcie z obszaru powiadomień wszystkich znajdujących się tam powiadomień;

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

Tabela 1. Atrybuty obiektu powiadomienia, potrzebne m.in. w momencie deklarowania nowego powiadomienia z użyciem funkcji schedule()

Atrybut	Typ	Opis
id	Liczba całkowita	Unikalny identyfikator, jednoznacznie identyfikujący powiadomienie. Domyślnie: 0
title	String	Tytuł powiadomienia, prezentowany w pierwszym rzędzie, jeśli powiadomienie zostanie wywołane. Domyślnie: pusty ciąg znaków w przypadku iOS lub nazwa aplikacji w przypadku Androida
text	String	Opis powiadomienia, prezentowany w drugim rzędzie informacji o powiadomieniu. Domyślnie: pusty ciąg znaków
every	String	Okres pomiędzy kolejnymi wywołaniami tego samego powiadomienia, jeśli ma się ono powtarzać. Może przyjmować wartości: „second", „minute", „hour", „day", „week", „month" lub „year". Domyślnie: 0, co oznacza, że powiadomienie jest jednorazowe i nie będzie powtarzane
at, firstAt	Data lub liczba całkowita	Data i czas, kiedy dane powiadomienie ma być po raz pierwszy wywołane. Jeśli będzie to data z przeszłości lub żadna wartość nie zostanie podana, powiadomienie zostanie wyzwolone natychmiastowo. Domyślnie: „now", co jest równoznaczne z new Date();
badge	Liczba całkowita	Etykietka w postaci liczby, która będzie prezentowana na ikonce aplikacji (w przypadku iOS) lub po prawej stronie powiadomienia (w przypadku Androida). Domyślnie: 0, co oznacza, że żadna liczba nie będzie pokazywana
sound	URI	Adres pliku zawierającego dźwięk przeznaczony do odtworzenia w przypadku wywołania powiadomienia. Domyślnie: res://platform_default
data	String	Ewentualne dodatkowe dane przechowywane w powiadomieniu w formacie ciągu znaków JSON. Domyślnie: null
icon	URI	Adres ikonki, która będzie prezentowana w obszarze powiadomień przy danym powiadomieniu; Opcja dostępna tylko w systemie Android. Domyślnie: res://icon
smallicon	URI	Adres małej ikonki, która będzie prezentowana w pasku powiadomień; Opcja dostępna tylko w systemie Android; Domyślnie: res://ic_popup_reminder
ongoing	Prawda lub fałsz	System Android umożliwia tworzenie powiadomień stałych, czyli takich, których nie można samodzielnie usunąć. Służą one np. sygnalizacji, że dana aplikacja jest włączona lub że trwa jakieś zdarzenie. Przykładem takiego powiadomienia jest wybór rodzaju klawiatury używanej w systemie, standardowo pojawiający się w obszarze powiadomień już od kilku ostatnich wersji Androida. Powiadomienia tego typu są prezentowane ponad zwykłymi powiadomieniami. Domyślnie: false
led	String	Kolor migania diody powiadomień, która zostanie uruchomiona, jeśli jest dostępna w danym urządzeniu (tylko na Androidzie). Wartość ta podawana jest jako ciąg znaków odpowiadający szesnastkowemu zapisowi trzech kolejnych składowych barwy (po dwa znaki na kolor) – czerwony, zielony i niebieski, czyli w tzw. formacie RGB. Np. FF0000 będzie odpowiadało kolorowi czerwonemu. Domyślnie: FFFFFF

- **cancel(id_powiadomienia, sukces)** – funkcja ta umożliwia usunięcie z harmonogramu powiadomień, które jeszcze nie zostały wyzwolone lub takich, które mają być powtarzane. W przypadku wywołania tej funkcji dla powiadomienia, które zostało już wyzwolone, zostaje ono usunięte z centrum powiadomień;
- **cancelAll(sukces)** – powoduje usunięcie z harmonogramu wszystkich zaplanowanych powiadomień;
- **getTriggeredIds(sukces)** – funkcja ta pozwala pobrać listę powiadomień, które są aktualnie prezentowane w centrum powiadomień;
- **getScheduledIds(sukces)** – funkcja ta pozwala pobrać listę powiadomień zapisanych w harmonogramie;
- **getAllIds(sukces)** – funkcja ta pozwala pobrać listę powiadomień zapisanych w harmonogramie lub wywołanych i znajdujących się w obszarze powiadomień; ta sama funkcja jest dostępna pod nazwą **getIds(sukces)**;
- **get(id_powiadomienia, sukces)** – funkcja pozwala pobrać jedno lub szereg (w przypadku podania tablicy identyfikatorów) powiadomień z harmonogramu lub centrum powiadomień;
- **getScheduled(id_powiadomienia, sukces)** – funkcja pozwala pobrać jedno lub szereg (w przypadku podania tablicy identyfikatorów) powiadomień z harmonogramu; w przypadku gdy żadne identyfikatory nie zostaną podane, a pierwszym parametrem będzie funkcja, pobrane zostaną wszystkie powiadomienia z harmonogramu;
- **getTriggered(id_powiadomienia, sukces)** – funkcja pozwala pobrać jedno lub szereg (w przypadku podania tablicy identyfikatorów) powiadomień z centrum powiadomień; w przypadku gdy żadne identyfikatory nie zostaną podane, a pierwszym parametrem będzie funkcja, pobrane zostaną wszystkie powiadomienia z centrum powiadomień;
- **getAll(sukces)** – funkcja pozwala pobrać wszystkie powiadomienia z harmonogramu lub centrum powiadomień;
- **isScheduled(id_powiadomienia, sukces)** – funkcja ta pozwala sprawdzić, czy powiadomienie o danym identyfikatorze znajduje się w harmonogramie; funkcja ta pomija powiadomienia znajdujące się w obszarze powiadomień, jeśli nie mają być powtórzone;
- **isTriggered(id_powiadomienia, sukces)** – funkcja ta pozwala sprawdzić, czy powiadomienie o danym identyfikatorze zostało wywołane i wciąż znajduje się w obszarze powiadomień systemu;

Tabela 2. Zdarzenia związane z powiadomieniami, których obsługę można zdefiniować	
Zdarzenie	**Opis**
schedule	Dodanie powiadomienia do harmonogramu
trigger	Wywołanie powiadomienia
update	Modyfikacja treści powiadomienia
clear	Usunięcie powiadomienia z obszaru powiadomień
clearall	Usunięcie wszystkich powiadomień z obszaru powiadomień
cancel	Usunięcie powiadomienia z harmonogramu
cancelall	Usunięcie wszystkich powiadomień z harmonogramu
click	Kliknięcie w powiadomienie

- **isPresent(id_powiadomienia, sukces)** – funkcja ta pozwala sprawdzić, czy powiadomienie o danym identyfikatorze znajduje się w harmonogramie lub w obszarze powiadomień;
- **on(zdarzenie, obsługa)** – funkcja pozwalająca na reagowanie na zdarzenia związane z powiadomieniami. W przypadku wystąpienia danego zdarzenia, wywołana zostanie funkcja obsługi, podana jako drugi parametr. Do każdego zdarzenia można przypisać kilka funkcji obsługi – nie ulegają one zastąpieniu.

W przypadku niektórych z powyższych funkcji, jako ostatni parametr można podać przestrzeń, dla której ograniczone ma być wywołanie. Najczęściej będzie to **this** lub np. **cordova.plugins**.

Dodawanie powiadomienia

W celu dodania do harmonogramu nowego powiadomienia, konieczne jest użycie wymienionej wcześniej funkcji **cordova.plugins.notification.local.schedule()**. Jej parametrem jest obiekt (lub tablica obiektów) o szeregu atrybutów, których opis przedstawiono w tabeli 1. Warto zauważyć, że część z atrybutów jest obsługiwana tylko i wyłącznie przez system Android, który jak zwykle, oferuje programistom najbardziej zaawansowane funkcje.

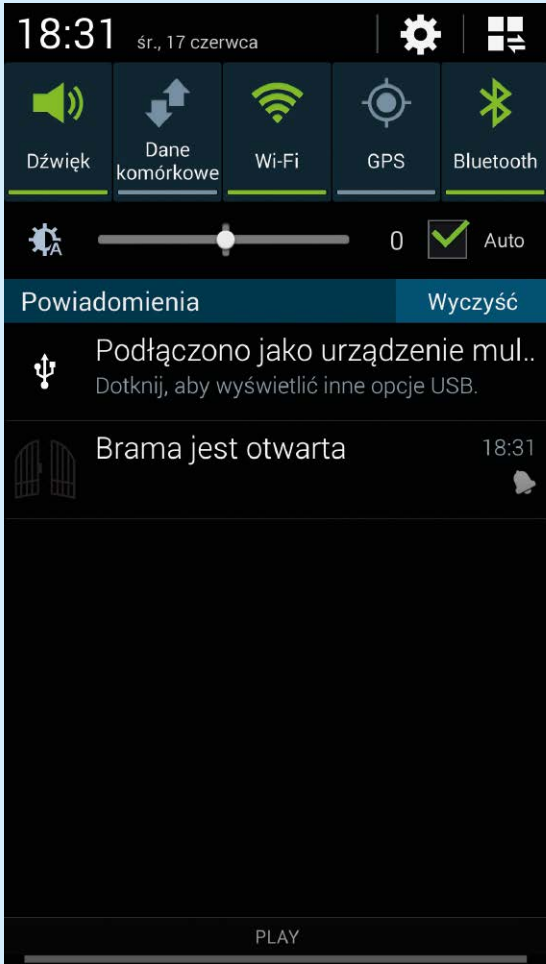
Dodanie powiadomienia następuje np. jak poniżej:

```
cordova.plugins.notification.local.schedule({
    id: 1,
    title: 'Powiadomienie z bramy',
    text: 'Brama została otwarta',
    sound: isAndroid ? 'file://sound.mp3' : 'file://beep.caf',
    icon: 'http://ep.com.pl/not_icon.png',
    data: { stan:1, czas: '2015-07-10 15:25:36' }
});
```

Warto zwrócić uwagę na format atrybutów takich jak **sound** czy **icon**, czyli tzw. URI (Uniform Resource Identifier). Dostępne są trzy rodzaje tego formatu:

- **http(s):** – obejmuje zasoby znajdujące się w Internecie i wymagające aktywnego połączenia internetowego, by je pozyskać; przykładowo może to być: 'http://ep.com.pl/not_icon.png';
- **file:** – obejmuje zasoby znajdujące się na lokalnym urządzeniu, a podana ścieżka jest określana względem folderu www projektu Cordovy; przykładowo może to być: 'file://sound.mp3'
- **res:** – obejmuje zasoby udostępniane poprzez ścieżkę systemową. Ścieżka jest określana względem katalogu **res/drawable/** w przypadku Androida i względem **Resources/** w przypadku systemu iOS; przykładowo może to być: 'res://sound.mp3'.

Zaraz po dodaniu powiadomienia (a można nawet wcześniej), warto przypisać do niego ewentualne



Rysunek 2. Powiadomienie w obszarze powiadomień systemu Android

funkcje obsługi zdarzeń, korzystając z funkcji **cordova.plugins.notification.local.on()**. Umożliwiają one reagowanie na takie zdarzenia, jak zaharmonogramowanie powiadomienia, usunięcie go, wywołanie, wyczyszczenie, modyfikację, czy przede wszystkim – kliknięcie przez użytkownika na dane powiadomienie. Lista dostępnych zdarzeń została zebrana w **tabeli 2**. Co więcej, jako parametry wywoływanej funkcji obsługi zdarzenia przekazywane są najczęściej dwa argumenty: identyfikator powiadomienia, którego dotyczy zdarzenie oraz stan pracy aplikacji, tj. informacja o tym, czy pracuje ona w tle, czy też jest aktywną aplikacją na ekranie urządzenia mobilnego. Wyjątek dotyczy zdarzeń takich jak „clearall" czy „cancelall", gdzie identyfikator powiadomienia nie jest przekazywany jako parametr. Warto dodać, że w przypadku zdarzeń występujących dla wielu powiadomień jednocześnie (np. gdy funkcji **cordova.plugins.notification.local.update()** podana zostanie tablica powiadomień jako argument), obsługa tych zdarzeń wywoływana jest wielokrotnie – po jednym razie dla każdego z powiadomień.

W trakcie dodawania powiadomień należy też pamiętać, że dodanie powiadomienia o identyfikatorze, który już istnieje spowoduje usunięcie starego powiadomienia i zastąpienie go nowym. Nie jest to równoznaczne operacji uaktualnienia powiadomienia funkcją **cordova.plugins.notification.local.update()**.

Powiadomienia tego typu są bardzo przydatne do tworzenia wszelkiego rodzaju przypomnień, np.

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

związanych z kalendarzami lub z odliczaniem czasu (korzystając z atrybutu **at**). Szczególnie wygodne może okazać się pole **data** przypomnienia, gdyż można w nim przechować dowolne dane, które następnie będą dostępne w momencie obsługi powiadomienia – czy to w przypadku jego kliknięcia czy usunięcia. Przykładowo, w polu tym można przechowywać informacje o akcji związanej z powiadomieniem – np. w naszym przypadku – rodzaj operacji (otwarcie lub zamknięcie bramy) i datę wystąpienia operacji.

Przykładowa aplikacja

Dla zademonstrowania możliwości płynących z wykorzystania powiadomień lokalnych, rozbudujemy aplikację pilota bramy, tak by w postaci powiadomień prezentował informacje o tym, że została otwarta lub zamknięta. System, po otrzymaniu informacji o otwarciu lub zamknięciu bramy będzie natychmiast generował powiadomienie. W wypadku, gdy brama zostanie otwarta przez dłuższy czas (np. minimum 5 minut), system będzie co pewien okres (np. co minutę) generował nowe powiadomienia, przypominające o konieczności zamknięcia bramy. Te będą odpowiednio umieszczane w harmonogramie po otrzymaniu powiadomienia o otwarciu bramy, a w przypadku otrzymania powiadomienia o zamknięciu, będą usuwane z harmonogramu.

Ponieważ utrzymamy w aplikacji funkcje zdalnego sterowania bramą, warto rozróżnić rozkaz otwarcia lub zamknięcia bramy, wydawany z pilota od potwierdzenia otwarcia (informacji o otwarciu) lub zamknięcia bramy, przesyłanego z napędu. To ten drugi komunikat, również przesyłany przez interfejs Bluetooth, będzie powodował generowanie powiadomień, a nie sam przycisk w aplikacji. W ten sposób brama będzie mogła być sterowana na różne sposoby (np. również bezpośrednio – ręcznie), a jej stan będzie cały czas monitorowany przez aplikację pilota, tak jak to pokazano na **rysunku 1**.

Trzeba też pamiętać, że harmonogram powiadomień jest prowadzony przez system operacyjny, a to oznacza, że jeśli jakieś powiadomienie zostało dodane do harmonogramu i ma się cyklicznie uruchamiać, to nawet jeśli aplikacja, która je zdefiniowała, została wyłączona, powiadomienie i tak będzie wywoływane zgodnie z harmonogramem. Co więcej, aplikacja Cordovy, która ma obsługiwać powiadomienia, będzie się automatycznie uruchamiała w tle, po włączeniu urządzenia mobilnego.

Na **listingach 2** i **3** pokazano fragmenty funkcji odpowiadających za obsługę powiadomień w omówionym przypadku. Naturalnie, w rzeczywistej aplikacji korzystniej byłoby dodawać do harmonogramu powiadomienie o identyfikatorze 3 (o tym, że brama jest wciąż otwarta) bezpośrednio po otrzymaniu komunikatu Bluetooth, a nie po wyzwoleniu powiadomienia o identyfikatorze 1 (na temat faktu otwarcia bramy), ale dla celów demonstracyjnych, skorzystaliśmy z obsługi zdarzenia „**trigger**". Równie dobrze można było też skorzystać ze zdarzenia „**schedule**", gdyż w naszym przypadku definiowaliśmy powiadomienia natychmiastowe.

Analizując kod z listingu 3 warto zwrócić uwagę na linijki:

var now = new Date().getTime();

*after_5minutes = new Date(now + 5 * 60000);*

które pobierają aktualną datę i tworzą zmienną **after_5minutes**, zawierającą aktualną datę, powiększoną o 5 minut (5 * 60 tysięcy milisekund).

Pojawiająca się na listingach 2 i 3 ikonka bramy (**gate.png**), aby była wyświetlana, została załadowana do odpowiedniego katalogu w projekcie aplikacji – w naszym wypadku do **c:\kursEP\pilot\platforms\android\res\drawable**. Wyświetlone powiadomienie zilustrowano na **rysunku 2**.

Listing 2. Fragment kodu odpowiadający za definiowanie powiadomień lokalnych w przykładzie z bramą. Pokazano tylko część związaną z odbieraniem danych przez Bluetooth. Reszta funkcji app.bt1() pozostaje niezmieniona, czyli taka jak na listingu 1.

```
bluetoothSerial.subscribeRawData(function(data)
{
    var bytes = new Uint8Array(data);
    if (bytes[0] == 1)
    {
        cordova.plugins.notification.local.cancel(2);
        var now = new Date().getTime();
        cordova.plugins.notification.local.schedule({
            id: 1,
            title: ‚Brama została otwarta’,
            icon: ‚file://gate.png’,
            data: {czas: now}
        });
    }
    else
    {
        cordova.plugins.notification.local.cancelAll();
        var now = new Date().getTime();
        cordova.plugins.notification.local.schedule({
            id: 2,
            title: ‚Brama została zamknięta’,
            icon: ‚file://gate.png’,
            data: {czas: now}
        });
    }
}, function()
    {
        alert(„niepowodzenie subskrypcji”);
    }
);
```

Listing 3. Dodatkowe polecenie, dodane do funkcji app.onDeviceReady(), definiujące obsługę zdarzeń związanych z powiadomieniami

```
cordova.plugins.notification.local.on(‚trigger’,
function(notification, state)
{
    if (notification.id == 1)
    {
        var now = new Date().getTime();
        after_5minutes = new Date(now + 5 * 60000);
        cordova.plugins.notification.local.schedule(
        {
            id: 3,
            title: ‚Brama jest otwarta’,
            icon: ‚file://gate.png’,
            at: after_5minutes,
            every: „minute”
        });
        cordova.plugins.notification.local.on(‚click’,
        function(notification, state)
        {
            if (notification.id == 3)
            alert(‚Wypadałoby zamknąć bramę!’);
        });
    }
});
```

Okienka dialogowe

Kolejnym elementem, przydatnym podczas generowania powiadomień dla użytkownika, jest mechanizm prezentacji systemowych okienek dialogowych. Będą one wyglądać różnie, w zależności od systemu operacyjnego, zainstalowanego na urządzeniu mobilnym, ale ze względu na przyzwyczajenia użytkowników, warto je stosować. Ponadto są dosyć wygodne w obsłudze. O ile w niektórych systemach operacyjnych okienka te będą miały postać przeglądarkowego okienka alert, takiego, jakie dotąd stosowaliśmy używając funkcji **alert()**, to w innych

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

Listing 4. Kod pliku index.html do przetestowania obsługi okien dialogowych

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="format-detection" content="telephone=no" />
        <meta name="msapplication-tap-highlight" content="no" />
        <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>
        <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 id="AlertButton"style="display:table;width:100%;height:100px;background-color:green;font-
size:xx-large;text-align:center">
            <div style="display:table-cell; vertical-align: middle;">ALERT</div>
        </div>
        <div id="ConfirmButton" 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;">CONFIRM</div>
        </div>
        <div id="PromptButton" 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;">PROMPT</div>
        </div>
        <div id="BeepButton" 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;">BEEP</div>
        </div>
    </body>
</html>
```

Listing 4. Kod pliku index.js do przetestowania obsługi okien dialogowych

```
var app = {
    initialize: function(){
            this.bindEvents();
    },
    bindEvents: function(){
        document.addEventListener(‚deviceready’, this.onDeviceReady, false);
    },
    onDeviceReady: function(){
        $(„#AlertButton”).click(function(){
            navigator.notification.alert(
                „nacisnąłeś przycisk Alert”,
                function(){
                    alert(„potwierdziłeś alert”);
                },
                „Powiadomienie”,
                „Potwierdzam”);
        });
        $(„#ConfirmButton”).click(function(){
            navigator.notification.confirm(
                „nacisnąłeś przycisk Confirm”,
                function(index)    {
                    alert(„wybrałeś opcję „ + index);
                },
                „Powiadomienie”, [„Potwierdzam”, „Anuluję”, „Nie wiem”]);
        });
        $(„#PromptButton”).click(function(){
            navigator.notification.prompt(
                „nacisnąłeś przycisk Prompt”,
                function(result)   {
                    alert(„wybrałeś opcję „ + result.buttonIndex + „ i wprowadziłeś wartość: „ + result.
input1);
                },
                „Powiadomienie”, [„Potwierdzam”, „Anuluję”, „Nie wiem”]);
        });
        $(„#BeepButton”).click(function(){
            navigator.notification.beep(2);
        });
    },
};

app.initialize();
```

przypadkach (np. w Androidzie) będą nieco bardziej rozbudowane i bardziej atrakcyjne.

Aby z nich skorzystać, użyjemy wtyczki **org.apache.cordova.dialogs**. Nasz dotychczasowy przykład zrobił się dosyć skomplikowany, więc pokażemy jak użyć okienek dialogowych na zupełnie nowym, czystym projekcie.

Tworzymy nowy projekt, dodajemy platformę i instalujemy wtyczkę poleceniami:

cd c:\KursEP
cordova create okna pl.com.ep.kurs Okna
cd okna
cordova platform add android
cordova plugin add org.apache.cordova.dialogs

Pobrała się wersja 0.3.0 wtyczki.

Poprawna instalacja wtyczki powoduje udostępnienie czterech metod:

- **navigator.notification.alert()** – okno najbardziej zbliżone do przeglądarkowego alertu, służące do wyświetlania powiadomień, które można jedynie potwierdzić;
- **navigator.notification.confirm()** – okno proszące użytkownika o wybór akcji. Pozwala na zdefiniowanie dwóch lub trzech różnych przycisków, a funkcja wywoływana po zamknięciu tego okienka dialogowa może pracować różnie, w zależności od wybranego przycisku;

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9

- **navigator.notification.prompt()** – okno proszące użytkownika o wprowadzenie danej i wybór akcji. Pozwala na zdefiniowanie dwóch lub trzech różnych przycisków, a funkcja wywoływana po zamknięciu tego okienka dialogowa może pracować różnie, w zależności od wybranego przycisku i oczywiście od podanej danej (ciągu znaków);
- **navigator.notification.beep()** – prosta funkcja umożliwiająca odtworzenie (wielokrotne) systemowego dźwięku powiadomienia.

Zanim użyjemy wtyczki, oczyszczamy pliki **index.html** i **index.js** ze zbędnego kodu oraz kopiujemy skrypt z biblioteką jQuery do podkatalogu **www\js** projektu. Tworzymy cztery duże przyciski „**AlertButton**", „**ConfirmButton**", „**PromptButton**" i „**BeepButton**", tak jak to robiliśmy w przypadku poprzednich przykładów w kursie oraz przypisujemy im odpowiednie akcje w funkcji **app.onDeviceReady()**.

Funkcja **navigator.notification.alert()** przyjmuje cztery kolejne parametry:

- **message** – ciąg znaków, zawierający wiadomość pokazywaną w oknie dialogowym,
- **alertCallback** – funkcja wywoływana po zamknięciu okna dialogowego,
- **title** – opcjonalny tytuł okna dialogowego; w przypadku braku tej zmiennej, tytułem będzie wyraz „Alert",
- **buttonName** – etykietka przycisku (ciąg znaków) w oknie dialogowym; domyślnie „OK".

Warto przy tym zaznaczyć, że funkcja **window.alert()** najczęściej powoduje wyświetlenie okna blokującego działanie aplikacji w tle, podczas gdy funkcja **navigator.notification.alert()** tworzy zazwyczaj okno nieblokujące, choć wiele zależy od systemu operacyjnego. W systemach gdzie brak jest standardowych okien dialogowych, można na stałe przypisać funkcję **navigator.notification.alert()** do wywołań funkcji **window.alert()**, dzięki czemu polecenie **alert()** będzie działać poprawnie. Można to zrobić pisząc kod:

```
window.alert = navigator.notification.alert;
```

Funkcja **navigator.notification.confirm()** przyjmuje cztery kolejne parametry:

- **message** – ciąg znaków, zawierający wiadomość pokazywaną w oknie dialogowym,
- **confirmCallback** – funkcja wywoływana po zamknięciu okna dialogowego, której przekazywany jest numer naciśniętego przycisku (1, 2, 3 lub wartość 0, gdy okno zostanie odrzucone – np. sprzętowym przyciskiem cofnij),
- **title** – opcjonalny tytuł okna dialogowego; w przypadku braku tej zmiennej, tytułem będzie wyraz „Confirm",
- **buttonLabels** – etykietki przycisków (tablica ciągów znaków) w oknie dialogowym; domyślnie dwie: [„OK",„Cancel"].

Funkcja **navigator.notification.prompt()** przyjmuje pięć kolejnych parametrów:

- **message** – ciąg znaków, zawierający wiadomość pokazywaną w oknie dialogowym,
- **promptCallback** – funkcja wywoływana po zamknięciu okna dialogowego, której przekazywany jest obiekt zawierający dwa atrybuty:
  - **buttonIndex** – numer naciśniętego przycisku (1, 2, 3 lub wartość 0, gdy okno zostanie odrzucone),
  - **input1** – ciąg znaków wprowadzony w polu w oknie dialogowym,
- **title** – opcjonalny tytuł okna dialogowego; w przypadku braku tej zmiennej, tytułem będzie wyraz „Prompt",
- **buttonLabels** – etykietki przycisków (tablica ciągów znaków) w oknie dialogowym; domyślnie dwie: [„OK",„Cancel"],
- **defaultText** – domyślny ciąg znaków, umieszczony w polu, w które użytkownik ma wprowadzić dane. Standardowo jest to pusty ciąg.

Funkcja **navigator.notification.beep()** przyjmuje tylko jeden parametr – liczbę naturalną, określającą ile razy ma być wywołany dźwięk powiadomienia. W przypadku gdy telefon jest wyciszony, dźwięk nie będzie wydawany, ani zastąpiony wibracjami.

Cały kod plików **index.html** i **index.js** projektu **Okna**, wykorzystujący opisane powyżej funkcje pluginu okien dialogowych został zebrany na **listingach 4** i **5**. Dla porównania, potwierdzenia naciśnięcia poszczególnych przycisków w prezentowanych oknach dialogowych wyświetlamy z użyciem klasycznego polecenia **alert()**. Porównanie wyglądu poszczególnych okien dialogowych w systemie Android 4.4 zostało zademonstrowane na **rysunku 3**.

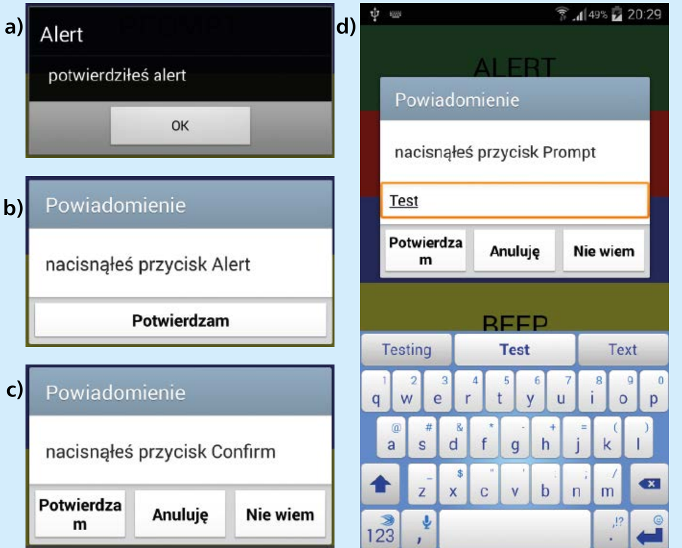
Powiadomienia Push

W niniejszej części kursu opisaliśmy powiadomienia lokalne i sposoby ich generowania, dodawania do harmonogramu oraz pokazaliśmy, jak nimi zarządzać. Kluczowe jest jednak to, że choć powiadomienia te mogą być uruchamiane w oparciu o sygnały pochodzące z zewnątrz – np. o komunikaty Bluetooth, to są to powiadomienia lokalne. By je wywołać, konieczne jest by na urządzeniu działała jakaś aplikacja i by samodzielnie decydowała o tym, kiedy wywołać powiadomienie lub by aktywnie nasłuchiwała na dane z otoczenia i w oparciu o nie definiowała powiadomienia. Istnieje jednak inny rodzaj powiadomień – Push – czyli takich, które są zdalnie przekazywane do urządzenia mobilnego.

Korzystanie z powiadomień typu Push to znacznie bardziej zaawansowany temat, a ich implementacja będzie się istotnie różnić pomiędzy poszczególnymi systemami operacyjnymi. Wynika to z faktu, że powiadomienia te są realizowane systemowo, w oparciu o zalogowanie się do sieci producenta systemu. Powiadomienia Push wysyłane są przez Internet do urządzeń, w oparciu o ich unikalne identyfikatory, których powiazanie z aktualnymi adresami sieciowymi (z IP) jest przechowywane na serwerach firm takich jak Google, Apple, czy BlackBerry. Aplikacja nie musi aktywnie nasłuchiwać poleceń Push, gdyż robi to za nią system operacyjny. Aplikacja jedynie musi zarejestrować możliwość otrzymywania określonych powiadomień z Internetu. Natomiast by nadać powiadomienie Push, jego nadawca musi nie tylko mieć zezwolenie odbiorcy na odbiór powiadomień, ale też musi być zarejestrowany na serwerach i w odpowiednich usługach poszczególnych twórców systemów opeoperacyjnych. Przykładowo są to:

- Google Cloud Messaging – dla systemu Android,
- Apple APNS Notifications – dla systemu iOS,
- Microsoft MPNS Notifications – dla systemu WP8,
- Microsoft WNS Notifications – dla systemu Windows 8.

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9



Rysunek 3. Porównanie różnych okien dialogowych: a) wywołanego poleceniem alert(), b) wywołanego poleceniem navigator.notification.alert(), c) wywołanego poleceniem navigator.notification.confirm(), d) wywołanego poleceniem navigator.notification.prompt(); warto zauważyć, że standardowe okienko alert() jako jedyne jest półprzezroczyste

Konieczne jest też odpowiednie zdefiniowanie uprawnień aplikacji, co wykracza poza zakres tej części kursu i dlatego temat powiadomień Push zostanie opisany w przyszłości, gdy pokażemy już, jak konfigurować dodatkowe ustawienia aplikacji.

Dodatek: zmiany w Cordovie

Na koniec tej części kursu wypada wspomnieć o zmianach, jakie zaszły w platformie Cordova od czasu rozpoczęcia powstawania tej serii artykułów. Apache Cordova to wciąż rozwijająca się platforma i dlatego przynajmniej raz w miesiącu pojawiają się jakieś aktualizacje związane z jej podstawowymi funkcjami, nie wspominając już o aktualizacji pluginów.

Cordova oferowana jest w postaci pakietu, któremu przypisane są konkretne wersje modułów. Przykładowo, Cordova 4.1.2, w której wykonywane są wszystkie zaprezentowane dotąd przykłady w kursie, ma przypisaną platformę Android w wersji 3.6.4 i to z jej użyciem dokonywano dotąd kompilacji. Już 6 lutego 2015 roku pojawiła się aktualizacja platformy Android do wersji 3.7.1, która m.in. podnosi domyślną wartość docelowego API androida kompilowanych programów do level 21, co podobno zwiększa szybkość prezentacji grafiki. Gdyby chcieć ręcznie zaktualizować konkretną platformę dla danego projektu Cordovy, należy wejść do katalogu, w którym projekt ten się znajduje i wywołać polecenie:

cordova platform update android@3.7.1

a gdyby tworząc nowy projekt chcieć od razu zainstalować nowszą wersję platformy, można wymusić zrezygnowanie z wersji 3.6.4 na rzecz 3.7.1 stosując polecenie:

cordova platform add android@3.7.1

Z aktualizacją pluginów jest o tyle łatwiej, że za każdym razem, gdy do danego projektu dodawany jest nowy plugin, jest on pobierany w najnowszej dostępnej wersji. Można więc odinstalować daną wtyczkę i zainstalować ją ponownie, by ją zaktualizować. W tym celu należy zastosować polecenia (przykład dla pluginu **org.apache.cordova.device**):

cordova plugin rm org.apache.cordova.device

cordova plugin add org.apache.cordova.device

Na początku marca ukazała się nowsza wersja Cordovy (4.3.0), a w połowie kwietnia pojawiła się kolejna wersja platformy androidowej – Cordova Android 4.0.0. Przynosi ona o tyle duże zmiany, że zawiera nowy interfejs WebView, oparty o projekt Crosswalk (plugin **cordova-plugin-crosswalk-webview**). Ponadto usunięto możliwość statycznego definiowania ekranów powitalnych w ustawieniach na rzecz konieczności stosowania nowej wtyczki **cordova-plugin-splashscreen**. Zmieniono też sposób określania uprawnień do uruchamiania wywołań do zewnętrznych serwerów. Konieczna stała się instalacja pluginów **cordova-plugin-whitelist**, który ma umożliwić wygodniejsze zarządzanie uprawnieniami.

Jak widać, zmienił się też sposób nazywania pluginów. Od wersji Cordova CLI 5.0.0 (udostępnionej w drugiej połowie kwietnia) identyfikatory w formacie **org.apache.cordova.*** zostają zmienione na **cordova-plugin-***. Zmiana ta związana jest ze zmianą repozytorium, w którym przechowywane są pluginy – będzie ono korzystało teraz z programu **npm**. Instalacja nowych pluginów będzie wyglądać podobnie jak dotąd, np. poprzez użycie polecenia (dla wtyczki **cordova-plugin-device**):

cordova plugin add cordova-plugin-device

Wyszukiwanie pluginów Cordovy z użyciem **npm** ma być możliwe dzięki atrybutowi **ecosystem:cordova**, który twórcy pluginów powinni umieszczać w swoich wtyczkach. Należy przy tym zaznaczyć, że dotychczasowe repozytorium pluginów, do którego odnosiliśmy się w przykładach w kursie będzie przynajmniej do 15 października tego roku, przy czym od 15 lipca będzie tylko w trybie do odczytu.

26 maja udostępniono aktualizację starszych wersji platformy Android dla Crodovy, ze względu na wykryty błąd. Wersję 3.7.1 zastępuje 3.7.2, a wersję 4.0.0 – 4.0.2.

Na początku czerwca pojawiła się też aktualizacja platformy Cordova Windows do wersji 4.0.0, w której zastąpiono obsługę systemu Windows 8.0 wersją 8.1 oraz wprowadzono obsługę Windows 10. Platforma ta wymaga posiadania Cordovy CLI w wersji 5.1.1 lub wyższej, również aktualnie dostępnej.

Podsumowanie

Dotychczasowa wiedza z kursu powinna pozwolić czytelnikom samodzielnie stworzyć prostą aplikację mobilną, szczególnie w przypadku kompilacji pod kątem systemu Android. Pokazaliśmy już, jak komunikować się na różne sposoby z otoczeniem, jak korzystać z niektórych funkcji sprzętowych oraz jak prezentować powiadomienia. W następnych częściach kursu będziemy chcieli zademonstrować, jak korzystać z pozostałych funkcji sprzętowych, jak debugować aplikację i jak ją kompilować na inne systemy operacyjne.

Marcin Karbowniczek, EP

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 10758, pass: 27qrg9k9