Exposing Native Device APIs to Web Apps

Arno Puder
San Francisco State University
Computer Science
Department
1600 Holloway Avenue
San Francisco, CA 94132
United States
arno@sfsu.edu

Nikolai Tillmann
Microsoft Research
One Microsoft Way
Redmond, WA 98052
United States
nikolait@microsoft.com

Michał Moskal
Microsoft Research
One Microsoft Way
Redmond, WA 98052
United States
micmo@microsoft.com

ABSTRACT

A recent survey among developers revealed that half plan to use HTML5 for mobile apps in the future. An earlier survey showed that access to native device APIs is the biggest shortcoming of HTML5 compared to native apps. Several different approaches exist to overcome this limitation, among them cross-compilation and packaging the HTML5 as a native app. In this paper we propose a novel approach by using a *device-local service* that runs on the smartphone and that acts as a gateway to the native layer for HTML5-based apps running inside the standard browser. WebSockets are used for bi-directional communication between the web apps and the device-local service. The service approach is a generalization of the packaging solution. In this paper we describe our approach and compare it with other popular ways to grant web apps access to the native API layer of the operating system.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Languages

Keywords

HTML5, Native API, Web apps

1. INTRODUCTION

Since the introduction of the iPhone in 2007, app stores have become a popular mechanism for hosting and distributing applications for mobile devices, so-called apps. Apps generated $53 billion revenue in 2012, and are predicted to generate $68 billion revenue in 2013 [22]. App development is much more constricted in the choice of technologies as every mobile platform favors different frameworks. Besides native apps that are written in the preferred programming language of the respective platform, HTML5 technologies gain traction for the development of mobile apps [12, 4].

A recent survey among developers revealed that more than half (52%) are using HTML5 technologies for developing mobile apps [22]. HTML5 technologies enable the reuse of the presentation layer and high-level logic across multiple platforms. However, an earlier survey [21] on cross-platform developer tools revealed that access to native device APIs is the biggest shortcoming of HTML5 compared to native apps. Several different approaches exist to overcome this limitation. Besides the development of native apps for specific platforms, popular approaches include cross-platform compilation [14] and packaging common HTML5 code into native apps [23].

In this paper we propose a novel approach that uses a generic *device-local service*, or *service* for short, that runs on the mobile device and that acts as a gateway, exposing native device APIs to HTML5-based web apps running inside a regular browser. We show how WebSockets and HTTP can be used for efficient bi-directional communication between web apps and the service. The service approach provides a clear separation between a web app, a web browser, and a device-local service, thereby generalizing the established packaging approach. By bundling the device-local service with the app it is also possible to mimic the packaging approach. Specifically, this paper makes the following contributions:

- a service-based approach to expose native APIs to web apps
- a reliable and efficient WebSocket-based communication protocol between the native shell and the web app
- an authentication and authorization scheme to address security and privacy concerns
- implementations for Android and Windows Phone

The paper is structured as follows: in Section 2 we present related work and provide a taxonomy by which the various approaches can be compared with each other. Section 3 explains in detail the device-local service approach proposed in this paper. Section 4 discusses implementations of the device-local service for the Android and the Windows Phone platform. Finally, in Section 5 we provide a conclusion and an outlook to future work.

MOBILESoft '14, June 2-3, 2014, Hyderabad, India

2. RELATED WORK

This section discusses various approaches that allow developers to use HTML5 for mobile apps. We distinguish between the platform, packaging, cross-compilation, and the device-local service approach. We provide a taxonomy to highlight the differentiating factors between existing approaches and our proposed solution.

2.1 Platform

One approach to expose native functionality to web apps is to add such support directly to the web app execution platform. That platform can be simply a web browser (e.g., Chrome under Android, Safari under iOS or Internet Explorer under Windows Phone) or it could be at the level of the operating system (e.g., WebOS, Tizen, Firefox OS).

The World Wide Web Consortium (W3C) coordinates the efforts to create general platform-level web APIs that should eventually be supported by browsers across all platforms. Mature W3C Working Drafts include specifications for accessing audio functionality [1], access to gyroscope, compass and accelerometer data [2], geo-location [17], media capture from microphone or camera [15], sending SMS, MMS and e-mail [7], full screen access [3], local database [16], two-way communication with a remote host [9].

However, even for many of those existing specifications, the actual implementation in most modern browsers is lagging behind and is not uniform across all browsers. For many other functions that can easily be accessed via native APIs on various mobile platforms, e.g. Bluetooth communication channels or Near Field-Communication (NFC), there are no mature W3C standards yet.

In some cases, W3C specifications make it more difficult for web apps than for native apps to achieve goal, for security or privacy reasons. For example, the Cross-Origin Resource Sharing (CORS) [20] specification requires server-side support for certain web requests to succeed. Native apps do not suffer from this restriction, and there is no proposed mechanism for web browsers to elevate web app permissions.

2.2 Packaging

The packaging approach bundles the HTML5 source of a web app including HTML, CSS, and JavaScript as data of a native app. The native app, that is written in the language of the respective platform, will instantiate a full-screen browser widget and load the bundled HTML5 into that widget (see Figure 1). The JavaScript running inside the browser widget can invoke functionality on the native side via some platform-specific mechanism. Likewise the native side can make up-calls to the JavaScript using a platform-specific mechanism. One benefit of the packaging approach is that the resulting apps can be hosted on the regular app store.

A prominent framework enabling the packaging approach is Apache Cordova (formerly PhoneGap) [23]. Cordova is available for all major mobile platforms. For each platform Cordova offers some boilerplate code written in the respective native programming language. Cordova features a plugin mechanism whereby support for new device API can easily be added. On Android, the boilerplate code is written in Java and makes use of the `WebView` widget. For down-calls, the method `WebView.shouldOverrideUrlLoading()` has to be overridden to intercept requests to URLs representing

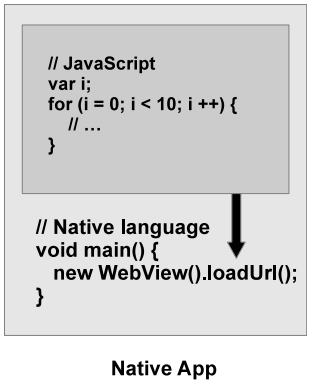

Figure 1: Packaging principle.

Cordova plugins. For up-calls, the Java code can use the method `loadUrl()` as shown in the following code excerpt:

```
1 // JavaScript calling Java
2 window.navigate("my-plugin://do-something");
3
4 // Java calling JavaScript
5 ((WebView) view).loadUrl("javascript:some_js();");
```

One major problem of the packaging approach is the fact that on most platforms the browser widget is not optimized in the same way the standalone browser app is. E.g., under Android versions earlier than 4.4, the `WebView` widget is not using any of the optimizations found in recent versions of the Chrome browser. Likewise under iOS, the performance of JavaScript in the `UIWebView` widget is several times slower compared to the Safari browser. While providing a cross-platform approach, Cordova applications are often visibly laggy due to restrictions of the platform.

Another problem of most implementations is the platform-specific interface between the native boilerplate app and the web app. This is usually realized via invoking JavaScript containing encoded messages. In cases when the messages are large (for example when transmitting an image from the phone picture library) this results in multiple encodings/decodings leading to high memory consumption and unreliability. As we show in Section 3.4, this is however not a limitation of the packaging approach itself.

While packaged apps typically ship with a particular version of the HTML, CSS, and JavaScript content, this is not a requirement. If the platform-specific interface is sufficiently generic, the web content could be updated dynamically.

2.3 Cross-Compilation

Cross-compilation is the process of translating one high-level language to another. The technique has gained prominence for mobile applications to facilitate cross-platform development while leveraging the SDK of the target platform. E.g., the XMLVM project [18] will cross-compile Android applications written in Java to Objective-C for iOS in such a way that the provisioning and signing can be done using Apple's Xcode. Starting from C# code, usually written for Windows and Windows Phone apps, Xamarin Studio[1]

[1] `http://xamarin.com/studio`

can cross-compile to create iOS, Android and Mac apps. In contrast to the packaging approach discussed in the previous section, cross-compilation yields better performance as the app is executed on the native layer and not in a heavyweight, poorly optimized web view widget.

The cross-compilation technique for web apps translates the JavaScript source code to the native layer. Because JavaScript supports features such as dynamic binding and prototyping, it is generally very difficult to translate JavaScript source directly to the static programming languages on most mobile platforms. Instead it is common to translate JavaScript to some intermediate representation such as byte codes whose execution generally achieves better performance compared to the native browser widget.

Appcelerator Titanium[2] is a tool for mobile development that uses cross-compilation. Apps are written in JavaScript and make use of Titanium specific APIs to interface with the native layer. The following code represents "Hello World" based on the Titanium framework:

```
// JavaScript using Titanium API
var win = Ti.UI.createWindow({
    backgroundColor : 'white'
});

var myLabel = Ti.UI.createLabel({
    text : 'Hello World',
    top : 250
});

win.add(myLabel);
win.open();
```

Titanium has its own UI abstraction layer for which it defines an API. It is noteworthy that UI elements such as the window or the label in the example above are mapped to native widgets of the target platform (and not to HTML as in the packaging approach), which gives a more native look-and-feel at the expense of lower cross-platform compatibility. During the cross-compilation process, Titanium statically analyzes the JavaScript source and builds a dependency list of all the Titanium APIs used by the application. A front-end compiler creates stubs code that includes appropriate platform-specific native code. Titanium also creates all necessary project files that allow compilation of the application using the native SDK of the target platform. On Android, the JavaScript is precompiled to byte code. At runtime, the byte code will be interpreted by the Rhino/V8 JavaScript interpreter [13, 5].

2.4 Device-Local Service

Accessing the native layer of a device via a service is a novel idea introduced in this paper. Instead of defining an API, access to a devices' capabilities is controlled by a protocol. The protocol is a generalization of the packaging approach. It allows the code that provides access to the native layer to reside in a separate device-local service that runs as a native app on the device. The web app runs in a standard browser and uses some IPC (Inter-Process Communication) mechanism to interact with the external service.

The app implementing the device-local service is generic and can be used by any web app. The native app that implements the device-local service can be hosted in the regular app store and needs to be downloaded and installed only once. The service listens on a local port and exchanges Protocol Data Units (PDUs) with the web app via WebSockets. WebSockets provide a portable, cross-platform way of communicating with the service. Special care must be taken to authenticate a WebSocket connection in order to avoid exploits by malicious websites and native apps.

An advantage of the service approach is that the web app runs in a standard web browser that offers much better performance compared to browser widgets. Although the service app has to be downloaded once from the app store, no further dependencies exist from a developer's point of view in terms of tools or SDKs. Special emphasis is given to the interoperability of the protocol: as long as the background service is available for a particular mobile device, the web app is agnostic of the platform it runs on.

By separating the application logic (web app) from the presentation layer (web browser) and the low-level device-specific implementation details (service protocol), the service approach supports versioning and security updates for each of those three components. Since the WebSocket-based protocol is a generalization of an API, the service can also be bundled with the app similar to the packaging approach. Section 3 will discuss all aspects of the device-local service approach in detail.

2.5 Comparison

Each of the four approaches (platform, packaging, cross-compilation, service) that offer native API to web apps have their advantages and disadvantages. They differ in their extensibility, hosting of apps, performance, as well as external dependencies. Extensibility refers to the ease with which new web API can be added. This is generally difficult for the platform approach as the complete platform needs to be updated, whereas other approaches are easier to augment. The various approaches discussed in this section also differ in the way web apps are hosted. Apps can either be hosted on the web or in an app store. Each has its benefits and drawbacks. App stores help with discoverability and generally inspire more trust in the apps (through screening processes and review systems). Web apps on the other hand are easier to deploy and update. Moreover, the app store content restrictions (for example on executing downloaded code) do not apply.

The approaches also differ in the performance of the resulting applications. The packaging solution offers an easy way to expose device API to a web app, however, since it leverages the native web view widget that is often poorly optimized, it generally results in low performance. The last distinguishing factor is the dependency of each approach to external tools used to build the application. The platform solution is clearly superior in this aspect, since the web API is already intrinsic to the platform and therefore does not require any special tools. The other approaches either have dependencies to an SDK that handles the compilation and packaging of the app or in case of the device-local service to a special app that needs to be downloaded and installed on the device. Table 1 gives an overview of the different characteristics of each of the approaches discussed in this section.

[2] http://www.appcelerator.com/

Table 1: Comparison of platform, packaging, cross-compilation and service approaches.

	Platform	Packaging	Compilation	Service
Extensibility	Difficult	Easy	Easy	Easy
Hosting	Web/App Store	App Store	App Store	Web/App Store
Performance	High	Low	High	High
Dependencies	None	SDK	SDK	Service

3. DEVICE-LOCAL SERVICE APPROACH

The device-local service approach advocates a protocol between the web app and a service that has access to the native layer of a device. The protocol is agnostic of the platform and can either be implemented as an external app hosting the service or packaged with a specific end-user app. The result is a clear separation between web apps and a service acting as a gateway to the native API.

This section gives a detailed description of the service approach. First we discuss the protocol itself in Section 3.1. In Section 3.2 we present an authentication and authorization framework. Section 3.3 explains the ownership rules when accessing resources stored on the device. In Section 3.4 we show how the device-local service approach can be used to mimic the packaging solution discussed in the previous section.

3.1 Protocol

The protocol between a web app and the service is defined by three individual components: (1) the transport mechanism, (2) the PDUs (Protocol Data Units), and (3) the order in which the PDUs are exchanged. In the following we give an overview of each of those components.

A web app needs a transport mechanism to interact with a service that is possibly implemented by a separate app. The transport mechanism must satisfy several requirements:

1. Communication must be bi-directional. In particular, it must be possible to push notifications from the service to the app.

2. Must support data exchange for large size resources such as image resources or music files. It should support the browser's asynchronous loading of these resources.

3. Can only use standard web technologies. This is necessary to break out of the browser's sandbox in a cross-platform way.

These three requirements are met by WebSockets and HTTP requests. The former can be used for efficient bi-directional exchange of (small) PDUs between the web app and the service. The latter is efficient for uploading or download large data chunks in the background.

The transport layer is used to exchange PDUs between the web app and the service. Request PDUs are sent from the web app to the service while response PDUs are sent in the opposite direction. We have opted to use JSON to describe PDUs. Each request PDU has at least the two attributes `action` and `id`. The former tells the service which action (i.e., API) is to be performed. The latter is used to match request and response PDUs. The response PDUs sent from the service back to the web app have at least the two attributes `status` and `id` where the `status` attribute provides an error code. Depending on the nature of the request or the response, additional parameters can be added via appropriate JSON attributes.

The following code excerpt shows how a web app can connect to the service via a WebSocket and then request gyroscope sensor readings:

```
1 // JavaScript
2 var ws = new WebSocket("ws://localhost:8042");
3 // Authentication & Authorization
4 var req = {action: "START_GYRO", id: 42};
5 ws.onmessage = function (e) {
6     var resp = JSON.parse(e.data);
7     if (resp.id == 42) {
8         // resp.x, resp.y, resp.z
9     }
10  };
11 ws.send(JSON.stringify(req));
```

The `START_GYRO` action will trigger a continuous stream of response PDUs that each contain the same ID (42) as well as the attributes x, y, and z for the actual sensor data. The web app can send the action `STOP_GYRO` if it no longer wishes to receive sensor data. While small-sized parameters (such as the sensor data in this example) are added as JSON attribute of the corresponding PDU, references to large resources are stored as URLs in the PDUs and the actual data is loaded by the browser via regular HTTP GET requests. Similarly, large resources such as images can be sent by the web app to the service via HTTP PUT requests.

The actual protocol is defined by the various actions that are sent by the web app. It is mostly a standard request/ response protocol where one request PDU results in one response PDU. The example of the gyroscope shows that one request PDU can also result in a sequence of response PDUs that is terminated by the `STOP_GYRO` action. In some cases, there is a finite set of response PDUs for a given action. E.g., the `LIST_CONTACTS` action will enumerate all contacts of a phone's contact database and send each contact in a separate response PDU. The web app can recognize the last response for a given request PDU via the JSON attribute `lastForId` that signals that this PDU will be the last for the request PDU with the matching id.

3.2 Security

While WebSockets provide an elegant and more importantly portable way of accessing an external service, it also leads to security risks. In some sense, the WebSocket pokes a hole through the browser's sandbox by giving access to potentially sensitive data stored on the phone. The user of a web app needs to be made aware of possible security threats

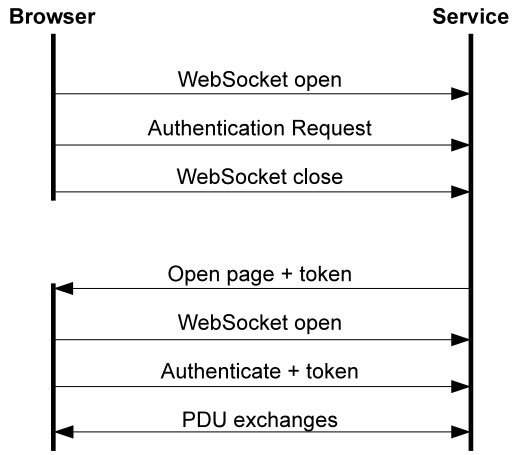

Figure 2: Authentication protocol.

and has to decide whether he or she is willing to give the web app certain permissions. Granting an app the permission to access the contact database could possibly lead to a privacy issue when those contacts are uploaded to a remote server. Ultimately the user has to trust a web app not to do anything harmful.

We adopt the Android permission model for mobile apps: an app has to request permission from the user when accessing sensitive resources. Permissions are requested via the special action `REQUEST_PERMISSIONS`. The granularity of the permissions is comparable to Android. Only once the user grants the permission, the web app can invoke the appropriate actions. For security reasons, the service and not the web app has to prompt the user for the permissions. Note that the service is assumed to be trusted and our security model does not address the case of a malicious service. The service will open a modal dialog and display the origin of the request along with the requested permissions that the user can either approve or deny.

In order for the user to make an informed decision about granting permissions, they have to know the actual origin of the request. The origin is communicated to the service as an HTTP header field when the WebSocket connection is established. If the connection is initiated by a trusted browser then the origin header field can also be trusted. However, it is also possible that a malicious app pretends to be a trusted browser and spoofs the header. We introduce a handshake protocol to disallow such spoofing.

Figure 2 shows the authentication protocol. First the web app opens a WebSocket connection to the service. The origin information of the HTTP header cannot be trusted by the service. The web app sends an authentication request and immediately closes the WebSocket again. During the next step, the service will use a platform-specific way to relaunch the web app via a trusted mechanism. In Android this trusted mechanism is the `ACTION_VIEW` Intent. The URI used as a parameter of the `ACTION_VIEW` Intent will include the domain of the authentication request plus a token that the service generates.

Once the web app has been re-launched, it will open a WebSocket to the service again. This time, it will authenticate the connection by sending the token back to the service. At this point the service can trust the origin of the request and PDUs can be exchanged via the WebSocket. Typically, after authenticating the web app will save the token in the browser local storage, so it can be reused later without relaunching. Our approach is similar to the implicit flow with a known redirection URI of the popular OAuth v2 protocol [6, 10].

3.3 Resource Management

WebSockets are particularly useful for bidirectional communication that facilitates pushing of notifications from the device-local service to the web app. Sensor data, such as the gyroscope mentioned earlier, can be pushed to the web app whenever new readings become available. However, the device-local service also provides access to components that return large payloads. Examples are the camera and the gallery app that return images or the media app that provides access to sound and video files. In principle it is possible to inline these resources within the response PDU via a so-called data URL [11]. The following JSON snippet shows the inlining of an image via a data URL:

```
// Response PDU, JSON
{
  "id": 43,
  "image": "..."
}
```

The Base64 encoded version of the binary image data is embedded in the JSON of the response PDU. Although data URLs allow transmission of arbitrary resources to the web app, this approach has distinct disadvantages. The complete resource needs to be encoded by the sender before it can be transmitted. Likewise, the receiver first needs to receive the resource in its entirety before it can be decoded, resulting in excessive memory consumption and visible lag in the web apps responsiveness. Browsers handle the loading of resources in a different thread and are usually capable of using a partially loaded resource. For example, a browser will typically start playing audio or video streams before they are fully transmitted, and progressively loading images will not block the UI thread.

For this reason the device-local service will also run a simple HTTP server that aides in the transfer of large volume payloads. The response PDU contains a URL that points to the HTTP server and the resource to be loaded. In order to secure the HTTP server from unauthorized access by other apps, the URL needs to include the same authentication token used to authenticate the WebSocket connection:

```
// Response PDU, JSON
{
  "id": 43,
  "image":
     "http://localhost:8044/example.png?AUTH=xyz"
}
```

The same origin policy of the web browser would normally disallow HTTP requests from the domain of the web app to `localhost:8044`. However, because we already have an authentication mechanism in place our HTTP server can safely disable this behavior using cross-origin resource sharing headers. Interestingly, no such headers are necessary in

the WebSockets server (which should just respond with 403 Forbidden, if it does not accept the origin of the request).

Although the embedded HTTP server solves the problem of efficiently loading resources lazily, it also introduces a new challenge. Since there is now an additional level of indirection – the URL points to a resource stored on the device – the question about the lifecycle of the resource arises. With such indirect resource URLs comes the need to properly manage the lifetime of the identified resources. Many resources represent objects which are already stored on the device, e.g., contacts, songs, pictures. As long as the objects are not removed from the device, the URLs will remain valid. Other resources are created at runtime, e.g., a still picture captured by the camera, or a microphone recording. For those resources, our convention is to add them to the respective local libraries (of pictures, sounds, etc.). It is then up to the user to delete these resources. It is expected that the web app will fail gracefully for resources that have been deleted by the user.

3.4 Optional Packaging

While the pure service approach provides a clear separation between a web app, a web browser, and a device-local service provider app, it is still possible to package everything together into one app. Packaging can be done in a way very similar to the Apache Cordova model discussed in Section 2.2, combining the HTML, CSS and JavaScript assets as data in a native app that instantiates a web view widget and also implements the service protocol discussed above.

The WebSocket server as well as the regular HTTP server that are embedded in the packaged app potentially expose their services via certain ports to any app running on the mobile device. In order to secure the server, the token-based authentication model mentioned in Section 3.2 is still needed. However, the steps of requesting permissions and relaunching the web app with a token can be simplified: When the packaged native app starts, it generates a random token and embeds it within the JavaScript code of the web app. When the JavaScript code connects to the service, it presents that token, and the service can trust the JavaScript code. This eliminates the need for the otherwise separate steps of requesting permissions involving a user interaction and relaunching the web app in order to provide it with a secret token. Instead, the permissions have to be granted when the packaged app is downloaded from the app store.

4. IMPLEMENTATIONS

We have implemented the device-local service approach for Android as an external app and via a packaging approach for Windows Phone, described in Sections 4.1 and 4.2 respectively. Section 4.3 details the API coverage of the current versions.

4.1 Android Implementation

In this section we show how the idea of a background service that acts as a bridge to a smartphone's native API was implemented for the Android platform. The project is dubbed WebAppBooster due to its features. Figure 3 shows the general architecture. We make use of an Android service that can run independently in the background on the device. The Android service launches a WebSocket server as well as a regular HTTP server that accept connections

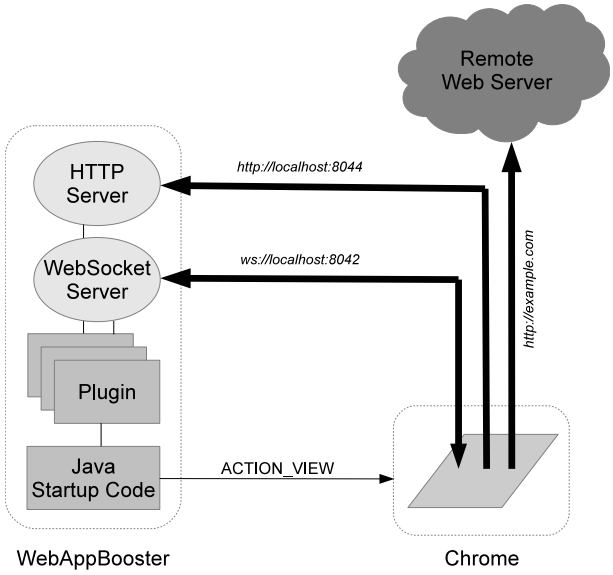

Figure 3: WebAppBooster architecture.

from localhost on different ports. The WebSocket server is used to exchange PDUs as explained in the previous section. The HTTP server is used to transfer large resources such as images or media files.

A browser such as Chrome or Firefox running on the Android device can be used to download a web app from a remote web site. The web app then uses a WebSocket connection to communicate with the Android service. WebAppBooster supports access to various native device API such as contacts, calendar, media as well as all sensors (camera, gyroscope, magnetometer, etc). WebAppBooster also provides access to the Bluetooth stack, allowing web apps to interact with Bluetooth-enabled devices. WebAppBooster is built using a flexible plugin mechanism that makes it easy to add new functionality.

As explained in the previous section, special attention needs to be given to the authentication mechanism. WebAppBooster needs to verify the origin of a request so that the user can rely on this information when deciding whether or not to grant permissions. To solve this problem, WebAppBooster will open a new trusted browser instance, using Android's Intent system. When WebAppBooster receives an authentication request from `http://example.com`, it will send an `ACTION_VIEW` intent with that URI. This intent is usually processed by a browser app, opening a new browser window and navigating to the given URI. If there happen to be multiple apps installed that can process the `ACTION_VIEW` intent, the user will be presented with a so-called activity chooser dialog. Even if a malicious app also registers an intent filter for `ACTION_VIEW`, it will not get launched automatically, but it will be up to the user to select the preferred target application.

Once a WebSocket connection has been authenticated, the web app needs to authorize the API it wishes to use. Authorization has to be handled by WebAppBooster as the trusted entity. This implies, that WebAppBooster needs to present the user with a modal dialog where the user can either accept or reject the request as can be seen in Figure 4. It is interesting to note that in the screenshot, the modal dialog was

Figure 4: WebAppBooster screenshot.

created by WebAppBooster and rendered over the Chrome browser that can still be seen in the background. It can be argued that allowing a background service to superimpose a user interface over the app running in the foreground is a security problem in Android. Since the Android API allows such usage, WebAppBooster makes use of this feature for its own purposes.

4.2 Windows Phone Implementation

On Windows Phone background tasks are generally only allowed to run periodically to conserve battery, and it is not possible to create a background service that is always listening for incoming requests. This prevented us from implementing a dedicated bridge service app as we did on Android with WebAppBooster. However, the `WebBrowser` control that can be embedded in regular apps on Windows Phone 8 is essentially identical to the built-in browser in terms of performance and capabilities. This allowed us to implement our service protocol using the packaging approach without the usual performance penalties.

A packaged app on Windows Phone consists of C# startup code, an embedded `WebBrowser` control, and an HTTP and WebSocket server which implement the service protocol via various plugins. Figure 5 shows the general architecture. On startup, the C# code issues a call `WebBrowser.Navigate(uri)` with a target `uri` such as `ms-appx:///index.html` to open the initial page of the embedded HTML code. The HTML5, CSS3 and JavaScript files used by the packaged app are exactly the same as the ones in the web app, with one exception: A randomly generated token gets embedded in the JavaScript code, which will later be used for authentication. The files reside in the so-called *isolated storage*, a location only accessible by the Windows Phone app itself. From within the `WebBrowser` control, the JavaScript code will then proceed to open a WebSocket connection at `ws://localhost:8042`. The JavaScript code uses the embedded token to authenticate (which is a simplified flow compared to the WebAppBooster implementation for Android where a security token gets communicated from the service to the browser by opening a new browser instance). The embedded HTTP and WebSocket servers, listening on localhost, will answer the request. The HTTP and WebSocket server have been implemented using only the Windows Phone C# SDK and the regular networking stack; they implement exactly the same protocol as our Android implementation.

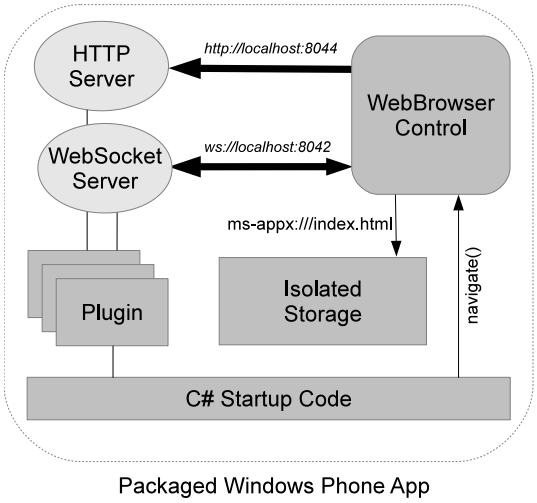

Figure 5: Windows Phone packaged app architecture.

The HTML5, CSS3, JavaScript and compiled C# binaries containing the startup and HTTP and WebSocket server code and plugins are all bundled together into a Windows Phone app.

4.3 Implemented APIs

We have so far implemented about 60 distinct actions (see Table 2). The device-local service offers capabilities to web apps that are usually limited to native apps. It provides access to all built-in sensors such as compass or gyroscope as well as device-local databases such as contacts or calendar entries. All of these are available in our Windows Phone implementation, and most of them are also available on Android. Some of them require further explanation:

- device orientation refers to both reading and locking it
- system-level notification refers to setting up a communication channel with the server, where the server can push notifications to the device, which are then displayed in the notification drawer (on Android) or as toast notifications (on Windows Phone)
- social sharing brings up platform-specific dialog for sharing content (links, images) on social media
- web request proxy lets the web app make arbitrary web requests circumventing the browser's same origin policy [20]

- local database provides a reliable local storage; while there are already two W3C standards for browser-based local databases [8, 16], we have found multiple issues working with them in practice: while Web SQL Database [8] is no longer on the W3C Recommendation track, it is the only database supported in (mobile) Safari at the time of writing. Indexed Database API [16] is on the W3C Recommendation track and supported by all other major browsers. However, all browsers impose low limits on how much can be stored (typically < 50MB), and when the limits are exceeded, they poorly communicate this problem to the user and the web app

Locking device orientation and taking screenshots may be difficult or impossible to implement sensibly with the service approach (the service does not know if the current running process is the browser with the web app in foreground). They are however very useful in the packaging approach.

The Android implementation of WebAppBooster is available as open source[3] and as an app in the Google Play Store[4]; a demo page on the web[5] enables Android users to explore the protocol.

One popular application that makes use of WebAppBooster in order to access sensors and other data is the TouchDevelop web app[6]. TouchDevelop is a general-purpose development environment that allows authoring and running mobile apps on mobile devices [19].

The Windows Phone implementation was used by the TouchDevelop team at Microsoft Research to publish the TouchDevelop app in the Windows Phone Store.[7] Five months after the release of the packaged TouchDevelop app for Windows Phone, the app was downloaded more than 80,000 times. It exposes most of the native APIs available on Windows Phone. In fact, the JavaScript code embedded in the packaged Windows Phone app is identical to the JavaScript code used in the web app, so all these services could get used on any other device in a web browser if a corresponding WebAppBooster implementation is available.

5. CONCLUSIONS AND OUTLOOK

This paper introduces a novel approach to grant HTML5-based apps access to the native layer of a device via the so-called *device-local service*. Instead of defining an API as in the packaging approach, the device-local service interacts with the web app via a protocol that allows for more flexible solutions. It can be implemented as a separate service running on the device or packaged with the app itself.

As part of the ongoing work we will add missing features of the service protocol to WebAppBooster. Future work includes the implementation of the service protocol for more platforms. While it would be technically possible on virtually all modern platforms, licensing restrictions of the stores used to distribute apps may not allow a service provider app that exposes local sensors and data. Recent iOS license changes would allow an app similar to the WebAppBooster app for Android.

[3] `http://sourceforge.net/projects/webappbooster/`
[4] `https://play.google.com/store/apps/details?id=org.webappbooster`
[5] `http://webappbooster.org/demo/`
[6] `https://www.touchdevelop.com/`
[7] `http://www.windowsphone.com/s?appId=fe08ccec-a360-e011-81d2-78e7d1fa76f8`

Table 2: List of services currently implemented for Android "A" and Windows Phone "WP". For each service we state the number of actions (#) and whether a special permission is required. Some services do not require a prior permission, but every action invocation is vetted by the user.

#	Category	Impl.	Perm.
1	Access to appointments	A/WP	yes
2	Access to contacts	A/WP	yes
5	Access to picture library	A/WP	yes
5	Access to music library	A/WP	yes
7	Access to music player	A/WP	yes
1	Playing sounds	WP	
1	Still picture capture	A/WP	yes
1	Recording from microphone	A/WP	yes
2	Reading accelerometer	A/WP	
2	Reading gyroscope	A/WP	
2	Reading compass	WP	
3	Device orientation	WP	
1	System-level notifications	A/WP	
1	Vibration control	A/WP	yes
5	Bluetooth connections	A/WP	yes
4	NFC tags reading/writing	WP	yes
1	FM radio control	WP	
1	Network information	A/WP	
1	Battery charge level	WP	
1	Sending SMS	WP	vetted
1	Speech recognition	WP	
2	Speech synthesis	A/WP	
1	Clipboard access	WP	yes
1	Social sharing	A/WP	vetted
1	Web request proxy	A/WP	yes
1	OAuth2 authentication	A/WP	
1	Taking screenshots	WP	yes
4	Local database implementation	WP	yes
1	Logging (for debugging)	WP	

The device-local service makes use of a WebSocket and HTTP servers running on the same device as the app itself. However, it might be an interesting idea to allow a web app access to the device-local service running on another device. This could be interesting for applications such as distributed storage and sensing networks.

Another interesting application is a "packaging service" that creates a native app by automatically packaging the web app with the device-local service. TouchDevelop already provides a packaging service to package JavaScript code into Windows Phone apps. Similarly, packaging services for other platforms could produce Android, iOS, or Windows Phone apps from web apps.

6. REFERENCES

[1] P. Adenot, C. Wilson, and C. Rogers. Web audio API. W3C working draft, W3C, Oct. 2013. http://www.w3.org/TR/2013/WD-webaudio-20131010/.

[2] S. Block and A. Popescu. Deviceorientation event specification. W3C working draft, W3C, Dec. 2011. http://www.w3.org/TR/2011/WD-orientation-event-20111201/.

[3] T. Çelik and A. van Kesteren. Fullscreen. W3C working draft, W3C, July 2012. http://www.w3.org/TR/2012/WD-fullscreen-20120703/.

[4] A. Charland and B. Leroux. Mobile application development: Web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.

[5] Google. V8 javascript engine. `http://code.google.com/p/v8/`. Accessed: 2014-01-22.

[6] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), Oct. 2012.

[7] D. Hazaël-Massieux, S. Chitturi, N. Widell, M. A. Oteo, and M. Froumentin. The messaging API. W3C working draft, W3C, Apr. 2011. http://www.w3.org/TR/2011/WD-messaging-api-20110414/.

[8] I. Hickson. Web SQL database. W3C note, W3C, Nov. 2010. http://www.w3.org/TR/2010/NOTE-webdatabase-20101118/.

[9] I. Hickson. The websocket API. Candidate recommendation, W3C, Sept. 2012. http://www.w3.org/TR/2012/CR-websockets-20120920/.

[10] B. Leiba. Oauth web authorization protocol. *IEEE Internet Computing*, 16(1):74–77, 2012.

[11] L. Masinter. The "data" URL scheme. RFC 2397 (Proposed Standard), Aug. 1998.

[12] T. Mikkonen and A. Taivalsaari. Apps vs. open web: The battle of the decade. In *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development (MSE'2011)*, pages 22–26, 2011.

[13] Mozilla.org. Rhino: Javascript for java. `http://www.mozilla.org/rhino/`. Accessed: 2014-01-22.

[14] J. Ohrt and V. Turau. Cross-platform development tools for smartphone applications. *Computer*, 45(9):72–79, 2012.

[15] I. Oksanen, A. Kostiainen, and D. Hazaël-Massieux. HTML media capture. Candidate recommendation, W3C, May 2013. http://www.w3.org/TR/2013/CR-html-media-capture-20130509/.

[16] J. Orlow, J. Bell, A. Popescu, E. Graff, J. Sicking, and N. Mehta. Indexed database API. Candidate recommendation, W3C, July 2013. http://www.w3.org/TR/2013/CR-IndexedDB-20130704/.

[17] A. Popescu. Geolocation API specification. W3C recommendation, W3C, Oct. 2013. http://www.w3.org/TR/2013/REC-geolocation-API-20131024/.

[18] A. Puder and O. Antebi. Cross-Compiling Android Applications to iOS and Windows Phone 7. *Mobile Networks and Applications*, 18(1):3–21, 2013.

[19] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 49–60, New York, NY, USA, 2011. ACM.

[20] A. van Kesteren. Cross-origin resource sharing. Candidate recommendation, W3C, Jan. 2013. http://www.w3.org/TR/2013/CR-cors-20130129/.

[21] VisionMobile. Cross-platform developer tools 2012. `http://www.visionmobile.com/product/cross-platform-developer-tools-2012/`, 2012. Accessed: 2013-12-02.

[22] VisionMobile. Developer economics Q3 2013: State of the developer nation. `http://www.visionmobile.com/product/developer-economics-q3-2013-state-of-the-\developer-nation/`, 2013. Accessed: 2013-12-02.

[23] J. Wargo. *PhoneGap Essentials: Building Cross-Platform Mobile Apps*. Pearson Education, 2012.