Towards the Implementation of a Push-Based Recommendation Architecture

Sergio Ilarri
University of Zaragoza, I3A
María de Luna, 1, Zaragoza (Spain)
Email: silarri@unizar.es

Manuel Herrero
University of Zaragoza
María de Luna, 1, Zaragoza (Spain)
Email: 681940@unizar.es

***Abstract*—Recommender systems play a key role for personalization. These systems help users when they need to choose among a variety of possible options, that may easily overwhelm them. In particular, in recent years, there has been increased interest in the development of recommender systems for mobile users. The so-called Context-Aware Recommender Systems (CARS) provide users with suitable suggestions by considering not only their preferences but also their specific context conditions.**

In this paper, we present our work in progress towards the development of a proof-of-concept implementation of a system architecture that can support the deployment of context-aware recommender systems for mobile users. We focus on scenarios where the recommendation process is not initiated by the user but by external entities that proactively trigger appropriate suggestions in different contexts.

I. Introduction

Recommender systems [1], [2] aid users to select suitable items (movies that they may like to watch, books to read, points of interest to visit, restaurants where they could have lunch, etc.) when the number of choices is large. Personalized recommender systems suggest items based on the preferences of each individual user, which are automatically learnt based on the information provided by the users (a typical feedback system is based on providing explicit ratings/scores). Moreover, it has been argued that, besides the user preferences, the context of the user is another important factor that should be considered when making recommendations. The intuition behind this idea is that context attributes can affect the relevance of certain items and types of items. Therefore, taking the context of the user into account will contribute to providing more suitable recommendations, which is important especially in scenarios of mobile computing, where the context of the user may change constantly. Thus, the traditional *User* $\times$ *Item* $\rightarrow$ *Rating* two-dimensional recommendation model has given rise to a *User* $\times$ *Item* $\times$ *Context* $\rightarrow$ *Rating* three-dimensional model. These novel types of recommender systems are called *Context-Aware Recommender Systems* (*CARS*) [3], [4], [5], [6].

In mobile computing environments, the context of a mobile user is highly dynamic: due to his/her movements, his/her location as well as the surrounding environment is changing constantly. Therefore, CARS could be particularly relevant in those types of scenarios. The location of the user is such a key context attribute that the specific term *Location-Aware Recommender Systems* (*LARS*) [7], [8] has been proposed to denote those recommender systems that focus on considering the relevance of the location of the user to provide more suitable recommendations. Besides, in a typical mobile computing setting, an approach where the user receives recommendations without the need of investing effort in explicitly asking for them, could be more appropriate. That is, *push-based recommendations* (proactive recommender systems) would be the preferred option, as opposed to *pull-based recommendations* (reactive recommender systems), as long as the recommendations provided are relevant.

In [9], we presented a theoretical framework for push-based recommendations in mobile environments. In this paper, we report our work-in-progress towards the development of a prototype for mobile devices, that could serve as a proof of concept of our proposal. The rest of the paper is structured as follows. First, in Section II, we provide an overview of the proposed push-based architecture. Second, in Section III, we summarize some alternative technologies that could be used for the development of a prototype. Then, in Section IV, we provide some details about the development of the prototype and the experience acquired. Finally, in Section V, we summarize our conclusions and the current status of the development.

II. Basics of the Push-Based Recommendation Architecture

From an abstract viewpoint, the push-based recommendation architecture presented in [9] is composed of the following elements:

- *Contexts*. In the proposed model, we call context to the scope or purpose of a recommendation (e.g., research, tourism, leisure, etc.).
- *Environments*. An environment is a physical or virtual area where a number of users can perform different types of activities and receive information concerning that specific environment. A user may belong to several environments at the same time. For example, a user in a mall in a city may belong to an environment defined by the limits of the commercial center and, at the same time, to an environment defined by the area of the whole city. Besides, the user may have logged into a specific website using his/her mobile phone, and therefore may also belong to an environment defined by that website.
- *Agents*. Agents are entities that generate (communicative and/or physical) *events* and they can be *users* and

Environment Managers (*EMs*). Users can be people receiving recommendations (e.g., a tourist in a city, a person in a shopping center, etc.) as well as special users acting on behalf of a business or organization and offering information about certain items or activities that can be performed (e.g., a shop announcing special offers). An EM is in charge of a specific environment and handles the membership of users within the environment and the communication within the environment.

- *Activities*. We use the term *activity* to denote a certain activity on an item. As opposed to simply *items*, in the architecture proposed in [9], we consider activities as the basis for recommendation (e.g., a system can recommend going to a specific restaurant for lunch or for dinner, rather than just recommend a restaurant).

In this paper, we focus on the software that will be executed on the mobile devices of the users and on the architecture of EMs.

III. Technologies Considered

In this section, we describe the technologies that we have considered so far in the development of our prototype. First, we provide an overview of the three types of approaches that can be followed for the development of mobile apps. Then, we focus on React Native. Afterwards, we describe some functionalities of Expo.

A. Native Apps vs. Multi-platform Solutions

According to IDC, the market share of operating systems for mobile devices in the first quarter of 2017 was as follows [10]: 85.0% for Android, 14.7% for iOS, 0.1% for Windows, and 0.1% for other operating systems. Even though the market is not as fragmented as a few years back (e.g., besides Google, Microsoft and Apple, companies such as Nokia and RIM were also popular in the market of mobile operating systems around the year 2010), there are still at least three mobile platforms that companies usually target, in order to reach as many mobile users as possible. For the development of mobile apps for this market, several options can be considered:

- *Develop native apps* (platform-specific development). With this approach, we would need to develop independent implementations for each target operating system (e.g., Android and iOS). This option usually leads to maximum flexibility and performance, as it directly exploits the resources of each platform, but it also implies a higher cost. Besides, it implies the need of managing several platform-dependent versions that need to be updated individually.
- *Develop a mobile web app*. In this case, the idea is to exploit the functionalities of standard web technologies and web browsers, usually available on any mobile device, to ensure the interoperability. By using technologies such as HTML5, CSS and JavaScript, we achieve the portability objective: only one mobile app needs to be developed, which can then be run virtually on any mobile platform.
- *Develop hybrid apps with the support of a framework*. Hybrid apps are built using frameworks such as *Adobe PhoneGap* [11], *Apache Cordova* [12], *Ionic* [13], *React Native* [14], *Xamarin* [15], and *Appcelerator* [16]. They exploit web technologies but at the same time provide access to platform-dependent functionalities, such as sensors, and offer a native look-and-feel for the graphical user interface.

Nowadays, the use of frameworks to develop hybrid apps seems to be the most popular choice, as it considers the trade-off between flexibility and development cost.

B. React Native

Finally, our chosen technology to develop the prototype was *React Native* [14], which is used by many popular apps [17], such as Facebook, Instagram, Skype, and Pinterest. React Native facilitates the development of native mobile apps using JavaScript [18] and React [19] (a component-based JavaScript library to build user interfaces that follows the "learn once, write anywhere" principle, which emphasizes the relevance of developing apps for each platform but using a common framework). In the following, we summarize some interesting benefits of React Native:

- *JavaScript-based to build mobile apps*. React Native allows to build native mobile apps using only JavaScript (and the React library). The mobile apps developed with React Native are native apps, rather than for example mobile web apps based on the use of HTML5.
- *Hot reloading*. It supports reloading the app without recompiling (new versions of the modified files are injected while the app is running), even retaining the application state if needed. This leads to a quick and convenient development.
- *Wrapping of native code*. If needed (e.g., for performance reasons), it is possible to write native code and execute it as part of the React Native app. This may complicate the maintainability of the project, due to the introduction of platform-specific components, but on the other hand enables fine-tuning it for each target operating system.

C. Expo

Besides, we initially considered also the use of *Expo* [20], which is a set of tools, libraries and services to build native iOS and Android apps by using JavaScript. Expo facilitates the development of React Native apps [21]. One of the most interesting features is the support for easy project management, as it can quickly push code updates to the mobile device with minimum effort from the developer and without the need to use specific development platforms such as Android Studio [22] (for Android devices) or Xcode [23] (for iOS devices). Besides, it provides several modules that offer interesting functionalities. Among those modules, we have tested and used the following ones in the first versions of the prototype:

- *Facebook* [24], which allows authenticating the user using his/her Facebook account. Once authenticated, with the access token obtained, it is possible to use the *Graph API* [25] developed by Facebook to retrieve (if allowed by the user) basic information about his/her profile (such as his/her city and date of birth). It should be noted that in the current prototype it would be possible to switch from an authentication system based on Facebook to another one. It might also be possible even to go without authentication, although a user identifier (e.g., composed by a local alias plus a unique identifier for the mobile device) would be needed in order to associate the ratings provided by the user to a user ID.

- *Calendar* [26], which allows interacting with the device's system calendars in order to retrieve and query information about events and reminders.
- *Location* [27], which allows retrieving the current location of the mobile device and it also offers the possibility to subscribe to receive location updates.
- *SQLite* [28], which allows querying a local SQLite database, which can be used to store information locally on the mobile device (e.g., recommended items and context variables).

These modules facilitate the access of different types of data that may be relevant to the recommender system. For example, querying the user's calendar is as simple as calling the method *Expo.Calendar.getEventsAsync(calendarIds, startDate, endDate)*, which retrieves all the events included in the calendars identified by the identifiers provided (*calendarIds*) and considering the specified time period ([*startDate, endDate*]). An inconvenience found is that the behavior of this method is not independent of the operating system used: on iOS, it returns all the events whose time span overlaps with the temporal interval [*startDate, endDate*] provided as an argument; on Android, it only returns the events that start on or after *startDate* and end on or earlier than *endDate*. So, if the app developed is executing on Android, it is not possible to directly obtain, in an efficient way, all the events that overlap with a given period of time.

IV. Development of a Prototype

In this section, we provide some details about the development of our prototype. First, we present an overview of the workflow of the app. Second, we describe the context data that are retrieved automatically by using sensors and information provided by other apps. Finally, we summarize some difficulties found with the use of background tasks.

A. Overall Workflow of the Prototype

An overview of the workflow considered in the prototype implemented is shown in Figure 1:

1) First, the mobile device of the user captures some context data, by exploiting different sources, like local sensors, information provided by installed applications, and/or external services that may offer information about the area.
2) Second, part of those user context data may be sent to the EM. It should be noted that the user can configure, through the mobile app, which data he/she is willing to share with the EM as well as the context data that he/she allows to share with the local app. For example, the user may allow retrieving information about his/her age from his/her Facebook profile, but maybe he/she does not want this information to be transmitted to any EM: in that case, the age is not sent to the EM but could be used by the app later as a post-filtering step.
3) Finally, the EM sends a set of recommended activities to the mobile device of the user. The app in the mobile device could apply a post-filtering by considering private attributes not communicated to the EM as well as other user preferences.

As shown in the figure, the communication between the mobile app executing on the mobile device of the user and the EM is performed through the exchange of JSON files, which is convenient given that it is a basic common structure well supported by React Native.

It should be noted that, according to the workflow presented, there are two basic components that need to be implemented: the mobile app, that executes on the mobile user device, and the EM architecture, that will probably be executed on a fixed computer. For the moment, we have focused our attention on the implementation of the functionality that runs on the mobile device. Nevertheless, we have also designed a minimal interface (RESTful API) that any EM's implementation should provide:

- {*@EM*}*/add/user/*{*userID*}*/*{*devToken*} *POST* adds a user to the environment of the EM specified; if the user is already registered, then potential changes related to his/her registration will be recorded. In the parameters, *@EM* represents the address of the EM receiving the request, *userID* is a user token that identifies the user (e.g., obtained through an authentication based on a Facebook account), and *devToken* is a token that identifies the mobile device of the user. For example, in an implementation with Expo the *devToken* could be an *expoToken*, and *Firebase Cloud Messaging* [29] also works with a token that identifies the device. If the EM rejects the registration in the environment, then a key-value pair, with an error code and its description, is returned to the user. A JSON file can be sent along with the call to this method to provide information that can be used by the EM to decide about the request (e.g., the location of the user).
- {*@EM*}*/delete/user/*{*userID*} *POST* removes a user from the environment of the EM specified.
- {*@EM*}*/event/user/*{*userID*}*/*{*length*} *POST* sends a JSON file to the EM with information about the current context of the user. The new *length* parameter is an optional parameter that indicates the maximum number of recommendations that the device wants to retrieve for a given situation. Upon calling this method, the EM could potentially trigger a recommendation process to try to find activities that are relevant in the context provided. The app executing on the user device could call this method of the EM periodically, by following the required context update policy.

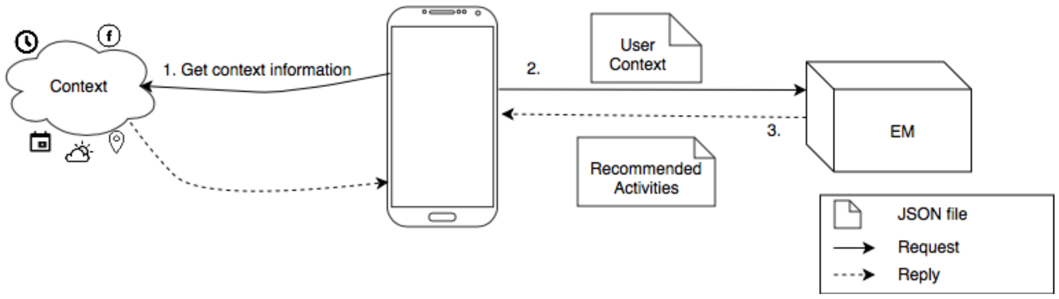

Fig. 1. General workflow in the implemented architecture

- *{@EM}/add/activity/ POST* adds an activity to the list of activities stored by the EM. This operation has not been defined for end users of the recommendation app; instead, it is constrained to be used by specific users (like administrators of EMs), that will be identified by the EM through an authentication token. The activity added is defined by a JSON file. Our prototype relies on a predefined catalog of types of activities, that is stored in a resource file associated to the app. If the catalog is expanded, a new version of the app with the resource file updated could be downloaded from the corresponding repository. Besides, if an EM defines a new own type of activity, it is added to a miscellaneous category ("Others"), which the developer of the mobile app could eventually integrate to the catalog, if there is a successful adoption of that type of activity by other EMs.
- *{@EM}/remove/activity/ POST* removes an activity from the list of activities stored by the EM. This operation acts as the counterpart of the previous one.
- *{@EM}/get/activity/{activityID} POST* obtains information about the activity identified by the parameter *activityID*, which should be in the list of activities stored by the EM.

By implementing this basic set of methods, any recommendation provider can participate and interact with the recommender mobile app running on the mobile device of the user. This structure offers high flexibility, as any provider is free to decide how to implement the services offered. We also envision the possibility of empowering the user with options in the app that enable him/her to opt in/out of the possibility of using certain EMs, which could be an effective measure to protect the user from potential abuses (like EMs flooding users with irrelevant information).

B. Context Data

In order to offer suitable recommendations to a mobile user, mobile CARS exploit context data to assess the potential relevance of different types of items/activities. Even though the user could be explicitly required to fill his/her context data when he/she provides a rating, by using a form, this can lead to a huge amount of attribute values not provided or left with default values (i.e., noise). For example, according to [30], the STS dataset [31], [32], which are data collected by the STS (South Tyrol Suggests) app, includes 14 context attributes but 89.37% of the context attribute values in the dataset are actually empty. Therefore, context data should be retrieved automatically, by using sensors of different types (physical, virtual, social, and/or human sensors [33]), if possible.

Currently, we are considering the following context data, as long as the user allows access to these data (see Listing 1 as an example of basic code that retrieves those data):

Listing 1. Basic code for retrieving context data when loading the app
```
async componentWillMount() {
   try {
      // Allow notifications
      myNotification._registerForPushNotifi-
          cationsAsync();
      // Get calendar events
      myCalendar._getCalendarAsync();
      // Get location and current weather
      this._getConditionsAsync();
      user = await AsyncStorage.getItem("user");
      token = await AsyncStorage.getItem("token");
      if (user !== null && token !== null) {
         // Go to Home
         this.props.navigation.navigate( "Home");
      } else {
         // No user available: go to Login
         this.props.navigation.navigate( "Login");
      }
   } catch (error) { // Error - repeat login
   this.props.navigation.navigate("Login");
}
```

- *Basic user profile attributes*. The mobile app can access the city and date of birth registered in the user's Facebook account (e.g., see Section III-C).
- *Location*. The GPS location of the mobile device is retrieved. For example, using the *Location* module of Expo (see Section III-C), the code shown in Listing 2 could be used.

Listing 2. Basic code for retrieving the current location
```
export async function _getLocationAsync() {
   try {
      // Ask the permission needed
      let status = await
          Permissions.askAsync(
          Permissions.LOCATION);
      if (status !== "granted") {
         throw "Permission to access the
             location was denied";
      }
      let location = await
          Location.getCurrentPositionA-
          sync({});
      // Save the location in the SQLite DB
      await AsyncStorage.setItem("location",
          JSON.stringify(location));
      await _getCurrentWeather(location);
   } catch (error) { console.log(error); }
}
```

- *Calendar data*. The calendars of the user are queried to obtain events that may affect the current date and/or time period (see Section III-C). Specifically, in our prototype, we are considering basic features of the events, like their time span (date/s and, if defined, also time), title, and description if available. Besides, we also obtain the name of the calendar where the event is registered, as it can reveal relevant information regarding the category of the event (e.g., the user may have calendars with names such as “Work”, “Leisure”, or even third-party calendars such as “National Festivities”). If the name of the calendar includes an email address, then the name is not retrieved, due to privacy concerns and the fact that an email address is usually not significant to characterize the event. An example code fragment is shown in Listing 3.

Listing 3. Basic code for retrieving calendar events for the next 7 days

```
export async function _getCalendarAsync() {
   try {
      // Ask the permission needed
      let {  status } = await
          Permissions.askAsync("calendar");
      if (status !== "granted") {
         throw "Permission to access
             calendar was denied";
      }
      // Gets an array of calendars
      let calendars = await
          Calendar.getCalendarsAsync();
      AsyncStorage.setItem("calendars",
          JSON.stringify(calendars));
      // Query the calendars
      let calendarIds = [];
      const end = new Date();
      const start = new Date();
      end.setDate(start.getDate() + 7);
      for (let index = 0; index <
          calendars.length; index++) {
         calendarIds.push(
             calendars[index].id + "");
      }
      let events = await
          Calendar.getEventsAsync(
          calendarIds, start, end);
      AsyncStorage.setItem("events",
          JSON.stringify(events));
    } catch (error) { console.log(error); }
}
```

- *Twitter data*. Trending topics in Twitter near the location of the user are obtained by using the Twitter API [34], thanks to the method *GET trends/place*, which retrieves the 50 trending topics for a specific WOEID (Yahoo! Where On Earth ID). To minimize the battery consumption of the mobile device and the amount of communications, and given that exploiting Twitter data may not be trivial, they can be obtained by the interested EM (rather than by the mobile device), in case information about the trending topics could be relevant to the recommendations.
- *Weather data*. Basic weather information in the current location of the user is obtained by using the OpenWeatherMap API [35] (see Listing 4).

Listing 4. Basic code for retrieving the current weather for a given location

```
export async function
    _getCurrentWeather(location) {
   let lat = location.coords.latitude;
   let lon = location.coords.longitude;

   try {
      let response = await fetch(
          "http://api.openweathermap.org/" +
          "data/2.5/weather" + "?lat=" + lat
          + "&lon=" + lon + "&appid=" +
          oauth.openweathermap);
      let resJson = await response.json();
      AsyncStorage.setItem("weather",
          resJson);
   } catch (error) {
      ...
   }
}
```

As commented before (see Section IV-A), the values of these context attributes are initially stored on the user’s mobile device and exploited locally. Additionally, if the user is willing to share these data, he/she can allow communicating them to the EMs, thus enabling not only a postfiltering of recommendations but also a prefiltering by the EMs themselves.

C. Difficulties Related to the Use of Background Tasks: Expo vs. React Native

Even though Expo boosts the development of mobile apps, thanks to the functionalities that it offers on top of React Native, we have found a major limitation. Particularly, the implementation of background tasks using that framework seems to be challenging. Running a task in the background would be needed in order to detect context changes, communicate them to the EMs where the user is currently active (if the user wants to share those context changes), and register the user into appropriate new environments as he/she moves around, even if the recommendation app is not executing on the foreground.

A possibility is the use of the library *react-native-background-task* [36], which supports the scheduling of a periodic task that is executed even if the app is executing on the background or even closed, as long as the execution period required is not higher than about 15 minutes. This means that, for example, a new environment could be detected with a delay of up to 15 minutes (i.e., if the user enters the limits of a new environment just after the background task has been executed for the last time and the user does not open the recommender app during that time lapse), even if the background task is programmed to be executed with the maximum frequency supported.

As an alternative, for the specific case of Android, *Headless JS* [37] could be used. The advantage of this library is that it does not impose any limitation regarding the frequency of execution of background tasks. The main disadvantage is that at the moment there is no port of Headless JS for iOS. Besides, it is not compatible with the use of Expo. Another existing library is *react-native-background-job* [38], which is based on React Native’s HeadlessJS, but it also implies a minimum execution period of 15 minutes.

Even though we started the implementation of the prototype with Expo, we have finally decided not to use it, due to the aforementioned difficulties. Instead, we now directly use React Native and Headless JS for the implementation of background tasks. Our preliminary tests show that Headless JS is a suitable approach for our problem. Therefore, we are currently re-implementing some parts of the code that relied on Expo's functionalities (mainly, tasks related to the capture of context data).

V. Conclusions and Future Work

In this paper, we have presented our current development of a prototype of a push-based recommendation architecture for mobile users. We have described the current status of the development along with some ideas about context data that could be exploited during the recommendation process by the EMs. Besides, we have mentioned several technologies that could be applied for the development. Our initial prototype was being developed using Expo, but now we are turning our attention to just using React Native, due to the limitations explained regarding the use of background tasks in Expo. The development of the prototype is still ongoing. Future immediate steps are refining and implementing the architecture of an EM, which includes defining the structure of the JSON files describing activities and the context of the user, among other tasks.

Acknowledgment

This work has been supported by the project TIN2016-78011-C4-3-R (AEI/FEDER, UE), the Government of Aragon - Aquitaine-Aragon project PASEO 1.0 (AQ-8), and DGA-FSE (COS2MOS research group).

References

[1] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender Systems Handbook*. Springer, 2011.

[2] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowledge-Based Systems*, vol. 46, pp. 109–132, 2013.

[3] G. Adomavicius and A. Tuzhilin, "Context-aware recommender systems," in *ACM Conference on Recommender Systems (RecSys)*. ACM, 2008, pp. 335–336.

[4] G. Adomavicius, B. Mobasher, F. Ricci, and A. Tuzhilin, "Context-aware recommender systems," *AI Magazine*, vol. 32, no. 3, pp. 67–80, 2011.

[5] Q. Liu, H. Ma, E. Chen, and H. Xiong, "A survey of context-aware mobile recommendations," *International Journal of Information Technology & Decision Making*, vol. 12, no. 1, pp. 139–172, 2013.

[6] M. del Carmen Rodríguez-Hernández and S. Ilarri, "Pull-based recommendations in mobile environments," *Computer Standards & Interfaces*, vol. 44, pp. 185–204, February 2016.

[7] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel, "LARS: A location-aware recommender system," in *28th International Conference on Data Engineering (ICDE)*. IEEE, April 2012, pp. 450–461.

[8] M. del Carmen Rodríguez-Hernández, S. Ilarri, R. Trillo-Lado, and R. Hermoso, "Location-aware recommendation systems: Where we are and where we recommend to go," in *International Workshop on Location-Aware Recommendations (LocalRec)*, vol. 1405. CEUR Workshop Proceedings, September 2015, pp. 1–8.

[9] R. Hermoso, S. Ilarri, R. Trillo-Lado, and M. del Carmen Rodríguez-Hernández, "Push-based recommendations in mobile computing using a multi-layer contextual approach," in *13th International Conference on Advances in Mobile Computing and Multimedia (MoMM)*. ACM, December 2015, pp. 149–158.

[10] IDC, "Smartphone OS," https://www.idc.com/promo/smartphone-market-share/os, May 2017, last Access: May 26, 2018.

[11] Adobe Systems Inc., "Adobe PhoneGap," https://phonegap.com/, last Access: May 26, 2018.

[12] The Apache Software Foundation, "Apache Cordova," https://cordova.apache.org/, last Access: May 26, 2018.

[13] Drifty Co. (d/b/a "Ionic"), "Ionic Framework," https://ionicframework.com/, last Access: May 26, 2018.

[14] Facebook Inc., "React Native – Build native mobile apps using JavaScript and React," https://facebook.github.io/react-native/, last Access: May 26, 2018.

[15] Xamarin Inc., "Xamarin," https://www.xamarin.com/, last Access: May 26, 2018.

[16] Axway, "Appcelerator," https://www.appcelerator.com/, last Access: May 26, 2018.

[17] Facebook Inc., "Who's using React Native?" https://facebook.github.io/react-native/showcase.html, last Access: May 26, 2018.

[18] Mozilla and individual contributors, "JavaScript guide," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide, last Access: May 26, 2018.

[19] Facebook Inc., "React – A JavaScript library for building user interfaces," https://reactjs.org/, last Access: May 26, 2018.

[20] 650 Industries, Inc., "Expo," https://expo.io/, last Access: May 26, 2018.

[21] ——, "Expo – frequently asked questions: What is the difference between Expo and React Native?" https://docs.expo.io/versions/latest/introduction/faq#what-is-the-difference-between-expo-and, last Access: May 26, 2018.

[22] Google LLC and Open Handset Alliance, "Android Studio," https://developer.android.com/studio/, last Access: May 26, 2018.

[23] Apple Inc., "Xcode," https://developer.apple.com/xcode/, last Access: May 26, 2018.

[24] 650 Industries, Inc., "Expo – Facebook module," https://docs.expo.io/versions/latest/sdk/facebook, last Access: May 26, 2018.

[25] Facebook Inc., "Graph API – user," https://developers.facebook.com/docs/graph-api/reference/v3.0/user, last Access: May 26, 2018.

[26] 650 Industries, Inc., "Expo – Calendar module," https://docs.expo.io/versions/v27.0.0/sdk/calendar, last Access: May 26, 2018.

[27] ——, "Expo – Location module," https://docs.expo.io/versions/v27.0.0/sdk/location, last Access: May 26, 2018.

[28] ——, "Expo – SQLite module," https://docs.expo.io/versions/v27.0.0/sdk/sqlite, last Access: May 26, 2018.

[29] Google, "Firebase Cloud Messaging," https://firebase.google.com/docs/cloud-messaging/send-message?hl=es-419, last Access: May 26, 2018.

[30] S. Ilarri, R. Trillo-Lado, and R. Hermoso, "Datasets for context-aware recommender systems: Current context and possible directions," in *First Workshop on Context in Analytics (CiA), in conjunction with the IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, April 2018, pp. 25–28.

[31] M. Braunhofer, M. Elahi, and F. Ricci, "Context-aware dataset: STS - South Tyrol Suggests mobile app data," 2013, DOI: 10.13140/RG.2.2.34480.97281.

[32] ——, "STS: A context-aware mobile recommender system for places of interest," in *22nd International Conference on User Modeling, Adaptation, and Personalization (UMAP)-Posters, Demos, Late-breaking Results and Workshop*. CEUR Workshop Proceedings, 2014, pp. 75–80.

[33] S. Ilarri, O. Wolfson, and T. Delot, "Collaborative sensing for urban transportation," *IEEE Data Engineering Bulletin*, vol. 37, no. 4, pp. 3–14, December 2014, special Issue on Urban Informatics.

[34] Twitter, Inc., "Twitter API – Get trends near a location," https://developer.twitter.com/en/docs/trends/trends-for-location/api-reference/get-trends-place.html, last Access: May 26, 2018.

[35] OpenWeather, "OpenWeatherMap API," https://openweathermap.org/api, last Access: May 26, 2018.

[36] J. Isaac, "react-native-background-task," https://github.com/jamesisaac/react-native-background-task, last Access: May 26, 2018.

[37] Facebook Inc., "Headless JS," https://facebook.github.io/react-native/docs/headless-js-android.html, last Access: May 26, 2018.

[38] V. Eriksson, "react-native-background-job," https://github.com/vikeri/react-native-background-job, last Access: May 26, 2018.