Android SDK

Integration Guide

SDK Version: 1.0.0

Date: 03/08/2013

Getting Started

This SDK is designed to be used by a developer building Android apps that will take advantage of the AppGlu Content, Insights, and Engagement features. To get started, you should already have an AppGlu account, with a new app created in the AppGlu Control Center. If you do not have an account yet, you can sign up at http://appglu.com, email sales@arctouch.com, or call 1-415-457-1111.

Introduction

The main goal of the SDK is to simplify the integration of your app with the AppGlu platform and leverage benefits like optimized data storage and synchronization, user data storage, app and content usage insights and smart handling of push notifications. It's designed with the platform specific best practices in mind, integrating with (and not replacing) the native APIs and frameworks whenever possible, hence reducing the need for learning new and vendor specific solutions and APIs.

The first step before using the SDK is to make sure you followed the integration steps specific to your development platform/IDE.

SDK Integration

The AppGlu SDK for Android devices requires Android 2.2 (API version 8) or above. It depends on a few external libraries that are included as part of the SDK zip bundle (under the libs directory).

The recommended way of integrating the SDK with your app is by extracting its distribution bundle and including all JAR files present on the libs folder to your classpath.

Before making any API calls the SDK has to be initialized with the corresponding API key/secret. The onCreate method of your application is usually the best place to initialize it.

```
// Production environment

AppGluSettings settings = new AppGluSettings("ENTER_APP_KEY", "ENTER_APP_SECRET");

AppGlu.initialize(this, settings);



// Staging environment

AppGluSettings settings = new AppGluSettings("ENTER_APP_KEY", "ENTER_APP_SECRET",
"staging");
AppGlu.initialize(this, settings);
```

Data Access

The AppGlu SDK provides different APIs for accessing and caching content from the AppGlu server, and for storing and retrieving user content back to the server.

The Content Sync Engine API provides the simplest approach by allows you to enable server-to-app syncing of content without changing your development approach. You use the frameworks (such as Core Data on iOS or SQLite on Android) that are considered best practice, and that you are probably already familiar with. The Content Sync Engine synchronizes the app's local storage with the AppGlu server behind the scenes. Content is updated, cached, and usable offline, without you having to worry about any of network requests or caching approaches.

The CRUD API is the best way to directly access AppGlu's server-side data, using the 4 standard CRUD operations: Create, Read, Update, Delete. The API handles the network connectivity for you, but does not do any specific caching to prepare and store the data for offline use.

The Saved Queries API can be used to run database queries that are created and stored by you on the server, allowing later customizations to the query without requiring app changes.

The User Data API is an easy way of storing user specific data and preferences.

The Content Sync Engine API: Data Synchronization

One of the most convenient functions of the AppGlu SDK is the data synchronization feature provided by the Content Sync Engine. It handles most of the work that developers would normally need to implement to fetch data stored on the server, parse, validate and cache the data for offline usage. The synchronization includes the data stored in the database tables as well as any resource files (such as images or movies) stored on AppGlu. Together, the data tables and secondary files represent the app content and when cached allows the app to be fully functional offline.

Data synchronization represents a big improvement in terms of flexibility and ease of use when compared to other approaches that require making one or more server requests per app screen to fetch remote content and relying on HTTP request based caching for offline support. With the Content Sync Engine API, you don't need to deal with the complexities of network requests (asynchronous operations, UI blocking and user feedback, data parsing, error handling, cache fallback) on every screen where content is displayed to the user. The content is fully available locally, and you are free to query the database and access the resource files directly from local storage without making any additional network requests. Offline support comes for free without any additional work.

The data synchronization is optimized for performance and optimal use of the network by incrementally transferring only modified data. This reduces battery drain and keeps data transfer usage to a minimum. Synchronization is for app data that is delivered from the server to the app, and should be considered read-only. Changes should not be made to the data locally, as those changes will not be synchronized back to the server. Instead, if you need to change data or store data on the server, you should use the CRUD API to make changes to synchronized app data, and/or the User Data API to store user data.

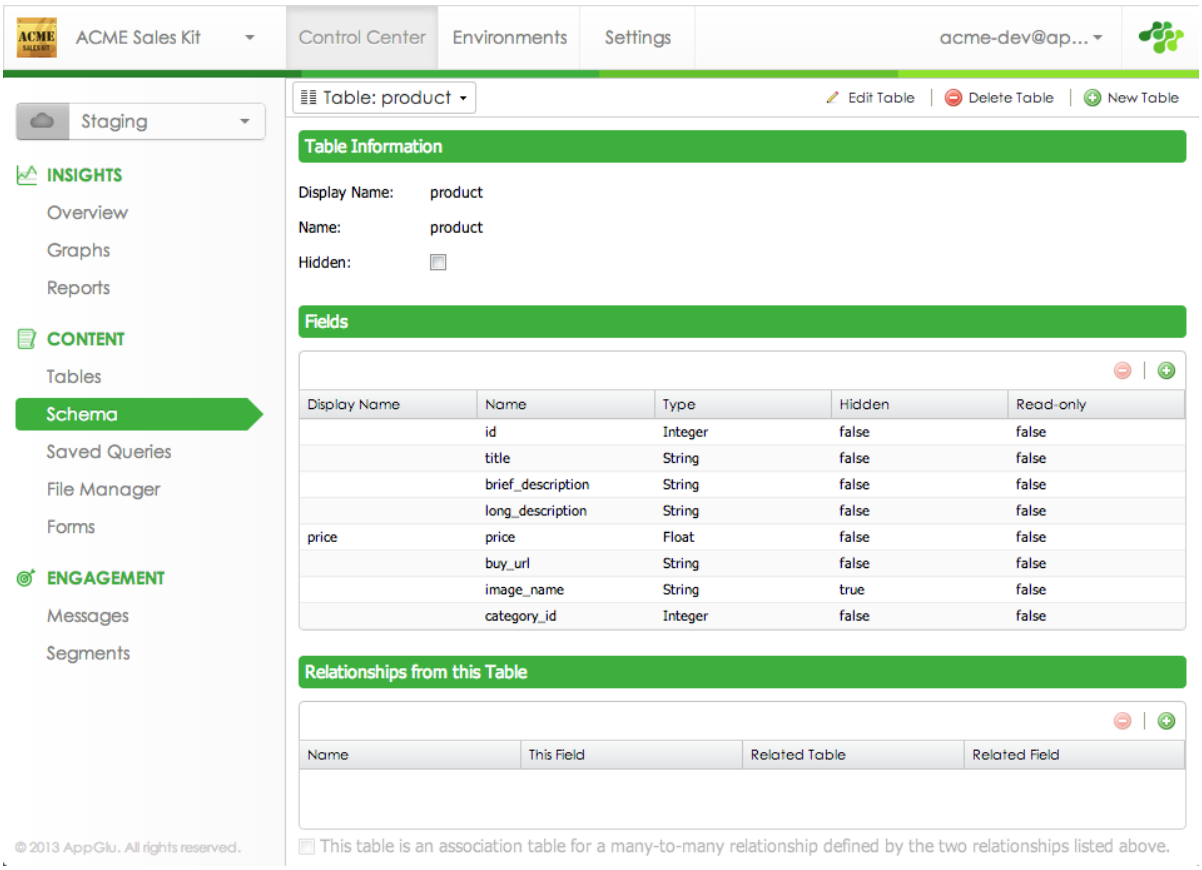
The synchronized app data gets stored in each platform's native storage system, and is accessible using the platform's native data access framework. Specifically, on iOS, the data is stored and accessible via Core Data, on Android the data is stored in SQLite, and for PhoneGap/HTML5 apps, the data is stored in Web SQL. Because the data is stored and accessed using the built-in native frameworks, there are no new steps to learn, and no new frameworks to implement. Synchronization and offline functionality is simple and automatic.

Synchronization with Android's SQLite

On Android the synchronization relies on SQLite for data storage. The SDK doesn't require changes to your existing database schema and you can continue to access it using the

same tools and frameworks you're used to (direct SQL access, ORM layers, DAO generators, etc). The Content Sync Engine API integrates directly with the SQLite database and synchronizes the data stored in AppGlu's server-side database tables.

The first step is to create the database schema inside of AppGlu and have it match the database schema of the app. You can do that using the Schema editor (screenshot below) inside of AppGlu's Control Center or through direct database access (see AppGlu's Control Center Settings tab for more information).



Once you have the schema created in the server side the next step is to have your custom subclass of SQLiteOpenHelper extend from SyncDatabaseHelper instead and rename the onCreate and onUpgrade methods to onCreateAppDatabase and onUpgradeAppDatabase, respectively. When the AppGlu SDK is initialized it's recommend that you pass an instance of your custom SQLiteOpenHelper, this will make it the default database to be used by the sync API:

```
AppGluSettings settings = new AppGluSettings(...);
settings.setDefaultSyncDatabaseHelper(new MyDatabaseHelper(this));
```

```
AppGlu.initialize(this, settings);
```

The synchronization engine relies on an Android Service to be executed and once started it continues to run even when no app activities are running. Services have to be declared in the app manifest file, so the SyncIntentService should be referenced this way:

```
<application ...>
	<service android:name="com.appglu.android.sync.SyncIntentService"/>
</application>
```

The SDK is now ready to start synchronizing the data by triggering the synchronization process as part of the app initialization flow. The exact moment where the synchronization starts depends on your specific app, but it usually happens when you main activity is initialized. To example below shows how to start the synchronization process:

```
SyncIntentServiceRequest request = SyncIntentServiceRequest.syncDatabaseAndFiles();
AppGlu.syncApi().startSyncIntentService(request);
```

The communication between the service that runs the synchronization and the other activities happens through broadcast messages:

```
private BroadcastReceiver syncBroadcastReceiver = new SyncBroadcastReceiver() {

	public void onPreExecute() {
	}
	public void onResult(boolean changesWereApplied) {
		loadImageExample();
	}

	public void onException(SyncExceptionWrapper exceptionWrapper) {

	}
	public void onNoInternetConnection() {
	}
	public void onFinish() {
	}
};


// Registering the sync broadcast messages receiver (inside of an Activity)

this.registerReceiver(syncBroadcastReceiver, new SyncIntentFilter());


// Unregistering the broadcast receiver
```

```
this.unregisterReceiver(syncBroadcastReceiver);
```

The example above includes the synchronization of all the database tables and the associated resource files. Check the reference documentation of SyncIntentServiceRequest to learn about other synchronization options.

The synchronized files are stored in the external device storage, so the app has to include the appropriate permission in its manifest file:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

To access the synchronized files a few different methods are provided in the SyncApi class. Below there are some examples of ways to access the files:

```
// Getting the content of an image file as a Bitmap

StorageFile storageFile = new StorageFile(myObject.image);

Bitmap bitmap = AppGlu.syncApi().readBitmapFromFileStorage(storageFile);


// Getting a file reference

StorageFile storageFile = new StorageFile(myObject.image);

File file = AppGlu.syncApi().readFileFromFileStorage(storageFile);


// Getting an InputStream reference

StorageFile storageFile = new StorageFile(myObject.image);

InputStream stream = AppGlu.syncApi().readInputStreamFromFileStorage(storageFile);
```

The CRUD API: Creating, Reading, Updating, and Deleting Data

When database table rows have to be added, modified, deleted or read directly from the server, the CRUD API should be used. It lets you perform the basic data operations in the app content tables.

As most of the AppGlu SDK APIs, the CRUD API methods are available in a synchronous and asynchronous versions. The table rows are represented as instances of the Row class, which is just a subclass of HashMap with additional helper methods to convert the data to the appropriate expected types (Integer, Float, Boolean, Date, etc). The examples below show how to use the CRUD API:

```
// Synchronously read "products" table row by ID and parse response as a Row
Row row = AppGlu.crudApi().read("products", "1");

// Asynchronously read "products" table row by ID and parse response as a Row
AppGlu.crudApi().readInBackground("products", "1", new AsyncCallback<Row>() {
       public void onResult(Row row) {
              ...
       }
});

// Read all "products" table rows
Rows result = AppGlu.crudApi().readAll("products");
Integer totalRows = result.getTotalRows();
List<Row> rows = result.getRows();

// Create new row in the "product" tables
Row product = ...;
AppGlu.crudApi().create("products", product);

// Update row in the "products" table
Row product = ...;
AppGlu.crudApi().update("products", "1", product);

// Delete row in the "products" tables
AppGlu.crudApi().delete("products", "1");
```

The Saved Queries API

Saved queries are a smart way of running database queries without requiring query logic to be hard coded in the app and allowing post app deployment customizations to the query. The query SQL gets stored on the server (under the Saved Queries section of the Control Center) and is reference in the app by its name. The Saved Queries API is used to run the query and

return fresh data from the server according to the logic defined in the query. Saved queries are the best way to return data that is composed by joining tables and at the same time allows later modifications to query logic and things like result display order.

```
// Get the list of top products defined by the "topProducts" saved query

QueryResult result = AppGlu.savedQueriesApi().runQuery("topProducts");

List<Rows> rows = result.getRows();


// Queries can also have input parameters

QueryParams params = new QueryParams();

params.put("category_id", "1");

QueryResult result = AppGlu.savedQueriesApi().runQuery("topProductsByCategory", params);

List<Rows> rows = result.getRows();
```

The Analytics API

The Analytics API lets you define the events and actions that are important to track and measure and which will later provide insights over user behavior and content usage. Having the right events tracked is very important to successfully measure the app effectiveness and drive future app and content updates.

Given the nature of Android apps, where activities are self-contained and independent of each other, some extra logic has to be put in place to properly capture the application life cycle and correctly track app level analytics. You have two options to implement that, either extending from one of the AppGlu analytics-ready activity superclasses (AppGluAnalyticsActivity, AppGluAnalyticsFragmentActivity, AppGluAnalyticsListActivity, etc) or doing it yourself by implementing the onActivityPause and onActivityResume methods of every activity in the app, like in the example below:

```
@Override

protected void onResume() {

	super.onResume();

	AppGlu.analyticsApi().onActivityResume(this);

}
```

```
@Override
protected void onPause() {
       super.onPause();
       AppGlu.analyticsApi().onActivityPause(this);
}
```

By initializing the Analytics API some events are automatically captured and metrics calculated, like the total number of users, duration and number of app sessions, number of active users, etc. In order to capture more detailed information about the actions the users take in the app and content that is more popular, the methods to log events should be called in the appropriate places of the app and giving as much context information as possible. Below are some examples of the different APIs to log events:

```
// Log an user action event
AppGlu.analyticsApi().logEvent("productListViewed"];


// Events can have parameters, giving context information about the action
Map<String, String> params = new HashMap<String, String>();
params.put("view_mode", "list_view");
AppGlu.analyticsApi().logEvent("productListViewed", params);
```

The Push Notifications API

Push notifications are the best way to keep users engaged, making sure they're always updated with the latest content and driving them back to the app. In order to measure the effectiveness of the push notifications it's important to track user's response to them.

Before your app can receive notifications you have to follow the steps to integrate Google Cloud Messaging with your app and configure the API keys in the AppGlu Control Center Settings. After that the app is ready to register for push when Google's API returns the registration ID. When the registration ID gets returned by the GCM registration service you just have to send it to AppGlu:

```
// Synchronously
AppGlu.pushApi().registerDevice(registrationId);
```

```
// Asynchronously
AppGlu.pushApi().registerDeviceInBackground(registrationId, new AsyncCallback<Row>() {
	public void onResult() {
	}
});
```

The User API: User Management and User Data

Analytics events and push notification tokens are stored anonymously, but apps that manage user identify, either through custom login systems or even social login, can use the User API to have the stored data associated to those users. This allows user targeted analytics and push messages and also simplifies the storage of user specific data.

The examples below show how to manage users and user data:

```
// Create a new user
User user = new User("usernane", "password");
AuthenticationResult result = AppGlu.userApi().singup(user);

// Login with existing user
AuthenticationResult result = AppGlu.userApi().login("username", "password");
Boolean userLoggedIn = result.succeed();

// Store data that is user specific, similarly to Shared Preferences
Map<String, Object> userData = ...
AppGlu.userApi().writeData(userData);

// Read the user data
Map<String, Object> userData = AppGlu.userApi().readData();
```