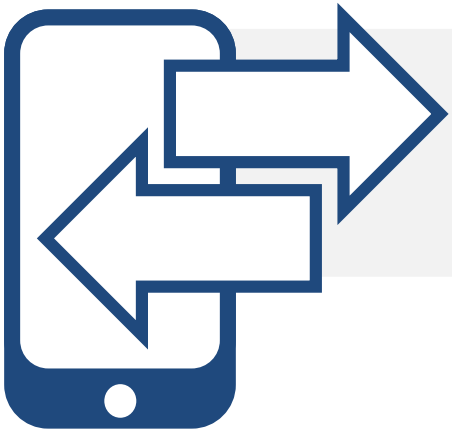
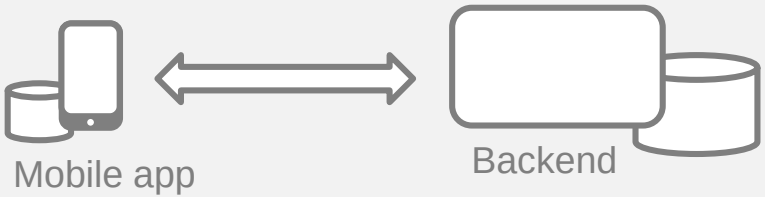


appSYNC



Open-source patterns & code for data synchronization in mobile apps

Sponsored by

TAPCROWD

v1.0, 24/07/14

Why appSync ?

Almost every mobile app needs to synchronize data with a server or backend (a.k.a. “the cloud” to make it more fancy), however it’s hard to find patterns, algorithms, strategies or example code to help developers to implement an optimal data synchronization strategy for the mobile apps that they develop. That’s why we started appSync: an open-source initiative around data synchronization for mobile apps, that helps developers with patterns, example code, and a naming standard. appSync can be used when developing a mobile app, or it can be integrated as part of an mBaaS solution (mobile backend as a service).

General definitions

Device or client: mobile device (smartphone or tablet) running an app that will synchronize data with a server/backend/cloud.

Server (or backend, cloud): centralized infrastructure that will synchronize data with multiple devices

Synchronization: sending data from devices to a server and vice versa, without loosing data and without unnecessary data transfer. Synchronization is especially needed in mobile apps that have a local database and that will sync data with a server when they are not offline.

Full synchronization: synchronizing all data (or all objects of the same type)

Incremental synchronization: synchronizing only data that has changed since the last sync

Object: a single data item that is synchronized between devices and a server, e.g. a database record, an agenda item, or a complex object including a customer and all his orders and invoices (e.g. in an ERP app).

Filtering: typically not all data is synchronized between a server and a device (e.g. data synchronized between the server and a given device can be limited based on the user account that is logged in).

Security and access control: out of scope for this project

Conflict handling: making a choose between two versions of an object, when the object changed on the server and on the client since the last sync.

General definitions

Primary key (PK): a property of the object that is unique across all objects of the same type, e.g. the property booking.data can be a PK when only one booking can be made per day.

PK conflict: a primary key conflict occurs when an object with the same PK is created on the server and on a device or on 2 different devices (e.g. a booking is created on 2 different devices for the same date, while only one booking is allowed per day).

GUID: a unique id, which is automatically generated (using random digits) for each new object. Guid's are guaranteed to be unique, even across devices and servers.

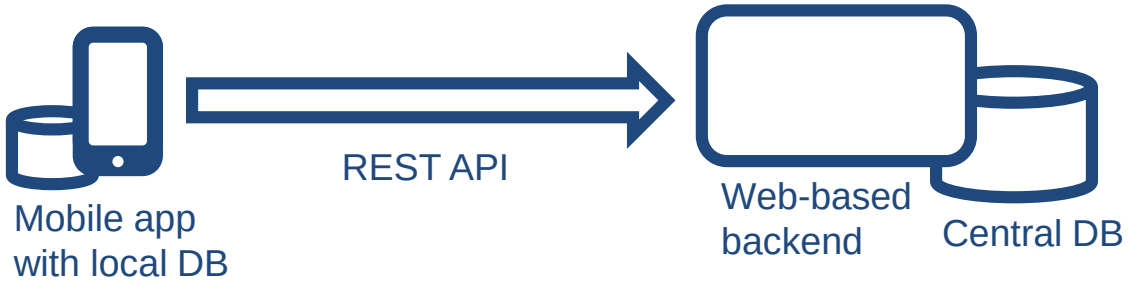
Counter: a concept introduced in appSync, to keep track of data changes on the server and each device. The counter is increased on each object change (new object, update object, delete object).

Soft delete: when an object is deleted, its property "isdeleted" is set to 1, but the object is not physically deleted in the database. This is considered safer because it allows for debugging and to undelete objects when needed.

Timestamp: for each object the timestamp (date & time) is stored from object creation (timestrampcreated) and the last object change (timestamplastupdate). Timestamps are not used to decide which data to sync because timestamps can be inaccurate (e.g. when the device clock is not identical to the server clock), but timestamps can be used to device which object version to keep when a conflict is handled.

The challenge of data synchronization

Why all the fuzz about data synchronization in mobile apps ? Let's take an example of a mobile booking app, and assume that you are part of the development team that will develop the app (for iOS, Android, Windows Phone and HTML5...), the REST API (for communication between the app and the server) and the backend. Our example booking app and backend will allow people to book rooms in a hotel. Our app and backend will have a typical architecture:



Maybe you will be using a commercial or open-source mBaaS solution and maybe that solution will provide data synchronization out of the box. In that case: great ! However, in many circumstances you will want to implement your own synchronization algorithm because of special requirements in terms of data availability, security, performance, bandwidth usage, data persistence etc. The goal of this project is to provide a basis that can be further adapted to your needs.

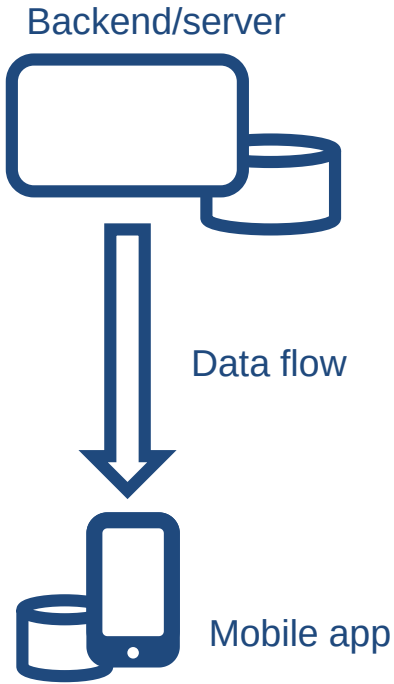
The challenge of data synchronization

The development team faces following challenges:

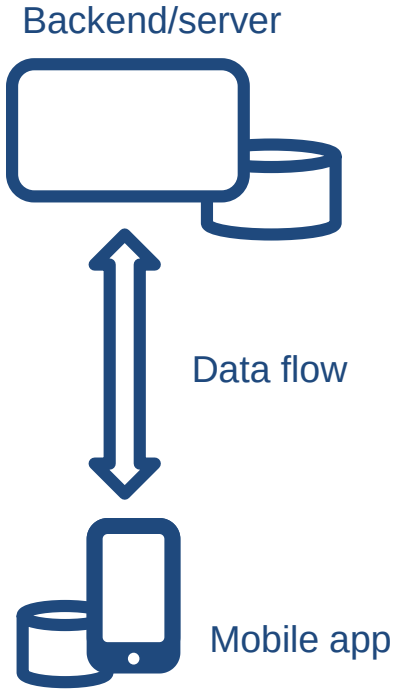
- Hotel rooms can be booked both in the web-based backend (on the server) and in the mobile apps. Bookings need to be synchronized as soon as possible in order to provide accurate availability information on the server and in the app on all devices.
- We do not want to send ALL bookings to the device on each change, this would be a huge overkill in bandwidth usage and data processing.
- A booking can be made on one device, and should then be available on all other devices as well (so syncing from a device to the server and then to all other devices is needed).
- Bookings can be updated or deleted on any device or on the server (regardless of where the booking was made). We risk that one given booking is changed on two devices or on the server and on a device between 2 sync jobs, this means we will have two different *versions* of the same object (booking). E.g. booking of room 100 on 12/05/2015 can have a price of 100 on device A and a price of 90 (discount is applied) on the server). These “conflicts” will have to be handled during sync.
- The app can be used offline, so we risk that the same booking is made on 2 devices (or on 1 device and on the server) between 2 sync jobs. Our primary key for objects of type “booking” consists of the room number + a date, which means we should only have one object for a given room on a given date. During synchronization we will have to handle “primary key conflicts”.

All these challanges will have to be solved in the synchronization pattern or algorithm that is used in our example booking project.

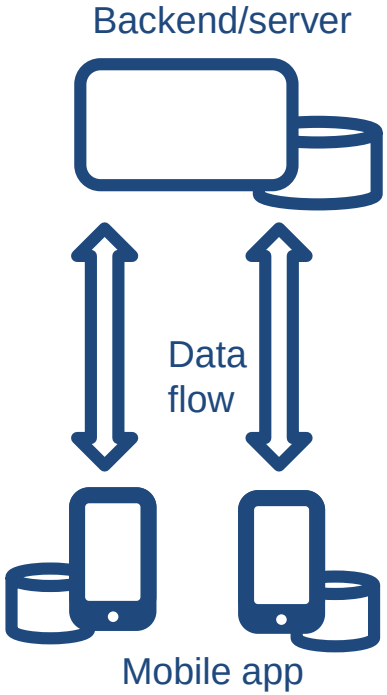
Synchronization types



1-way sync



2-way sync



Multi-way sync

Synchronization types

One-way sync: data is only synced from the server to the apps (e.g. news app where content is synced from the backend CMS to the apps) or data is synced from the apps to a server (e.g. logging/analytics).

Two-way sync: data is synced in two directions, from an app to a backend and back. E.g. a user is logged in and can manage his own data on a website and in an app (assuming user cannot be logged in on 2 devices at the same time).

Multi-way sync: data is synced from multiple devices to a server/backend and back. This also means that data from one device is synced to the server and from the server to other devices (e.g. collaboration apps...). This is the type of sync that is covered with the appSync code.

Assumptions made in appSync example code

1. Mobile app (device) always initiates a sync, not the server. The reason is that the app knows best when is a good time to sync data, e.g. based on connectivity or user actions in the app.
2. The mobile app handles conflicts, not the server. This means that conflict handling code resides on the device. The reason is that in many cases a conflict requires informing the user (e.g. “sorry, your recent booking cannot be confirmed, this room is already booked”). Note: this does NOT mean that the object version on the device automatically takes priority ! Which object version “wins” (server or client) depends on the conflict handling strategy that you have chosen for your project.
3. Mobile app always syncs from server to app first (“downloading” new data), and then from app to server (“uploading” new data).
4. The server is not aware of the sync status of the clients, the clients must each keep track of which data they have already synced and which not. This is because mobile apps (clients) can be deleted, re-installed etc. and the server should not keep track of all this activity.
5. Transactional consistency is currently not implemented in the appSync example code. This means the code does not guarantee full data consistency. This can be solved by extended the appSync code with locking concepts, and/or by executing a “full sync” on regular intervals, in between “normal incremental” syncs.
6. The appSync code currently does not handle data dependencies, no specific order is used to sync objects. In case objects have dependencies (e.g. parent/child with foreign keys), these must be custom implemented in the code.

appSync multi-way sync algorithm explained

Every object has a unique guid. When a device or the server receives an object with a new (unknown) guid, it will create a new object in its local database. In case the guid is already known, the object in the local database will be updated (e.g. its “value” parameter).

Each device and the server have their own counter, the counter is increased on each change of an object in the local database, so each time a new object is created or an object is updated or deleted. The counter is a point in time, and it is used to keep track of which data has already been synced.

Note: the counter is not increased on the client, when an object is updated due to the sync process, only when an object is changed directly on the client (e.g. user creates a new booking in a booking app). On the server however, the counter is also increased when an object is changed due to the sync process, because this object change must be synced to other clients as well. In other words: the counter must be increased indicating that the server data has changed and making sure that other clients will fetch new data on their next sync.

appSync multi-way sync algorithm explained

Each device (client) will keep track of its sync status (meaning the data it has already synced to the server and from the server) using these 2 parameters:

- Counter_lastsync: value of the local counter of the client on the last sync TO the server (upload)
- Servercounter_lastsync: value of the counter on the server, on the last sync FROM the server (download)

Note: the server itself does NOT have these 2 parameters (only the clients), because the server does not keep track of the sync status of all the clients (see rule 4 above).

Each object has a property counter_lastupdate, both on the server and on the clients. This is the value of the local counter when the last update to the object was done (creation of the object, an update or deletion).

Each client (the mobile app) must decide when to sync with the server, e.g. after launching the app, after user login, when a user clicks “refresh”, every minute, on each local update of data...

The app will sync by calling a method doSync(). This method will call syncFromServer() first and then syncToServer(). This means that the apps will always get new data from the server first (this is to be able to handle conflicts) and then send new data to the server. This is an important aspect of the whole appSync code, and the code will break (resulting in potential data loss during sync) if this rule is not honored !

appSync multi-way sync algorithm explained

The method syncFromServer() on the client, will ask the server for all data changes since servercounter_lastsync.

The server will answer this request by fetching all objects with object.counter_lastupdate > servercounter_lastsync and sending them to the client.

The server will also send its current counter to the client. The client will store this new value in servercounter_lastsync (because this is the server counter for which the client has received all data changes).

Next, the method syncToServer() on the client, will send all its data changes to the server since counter_lastsync. The client will fetch its objects with object.counter_lastupdate > counter_lastsync and send them to the server.

When the server confirms have received the data (result = ok), the client will update its counter_lastsync with the current value of its counter.

This completes an incremental sync cycle.

In case the client wants to execute a full sync, it simply has to reset its servercounter_lastsync and counter_lastsync to zero, and then execute a sync cycle as defined above.

Conflict handling strategies

In the above sync algorithm, when the client receives data from the server, the client will update its local objects or create new objects in its database based on the guid of each object. It is however possible that the client receives an update of an object from the server, which is also updated locally on the client since the last sync (e.g. last sync was 2 hours ago, and since then object with guid 12345 has been updated both on the server and on a client). In this case the client has two versions of the object (one from the server, and one local version) and it has to choose which version to keep. This means the client has to handle a conflict.

A conflict occurs when the server sends an updated object and on the client object.counter_lastupdate > counter_lastsync (which means object update is more recent than last sync from client to server, or in other words server did not yet receive client update for the object).

The appSync code allows to choose between 3 types of conflict handing:

Server priority: in case of conflict, the client will choose the object version from the server and thus overwrite its own last update of the object. Typically the user of the app will have to be informed of this overwriting (e.g. “your booking has been cancelled”).

Client priority: in case of conflict, the client will choose its own object version, and disregard the version from the server. In the second phase of the sync, the client will send its own version of the object to the server and the server will update its own object with this new version (meaning the client “wins”).

Timestamp priority: in case of conflict, the object version with the most recent timestamp (timestamp_lastupdate) will be kept, this version will be stored on the client and synced to the server as well.

Primary key (PK) conflicts

The guid is a guaranteed unique identifier. Next to the guid, the objects may also have a primary key (PK), which consists of one property or a combination of properties (e.g. room number + date for objects of type “booking”). When the server creates an object with a certain PK, and a client creates another object with the same PK, then a PK conflict will occur on the client during syncing.

In the example of our booking app, this would mean that the same room is booked on the same date on the server and on the client (possibly for two different customers, meaning an overbooking which must be solved).

The same occurs when two different clients create objects with the same PK: the first client A will sync the object to the server. The second client B will receive this object when syncing from the server, and will handle the PK conflict. It will then sync to the server. In a second iteration, client A will sync from the server, and receive the updated object and also handle the PK conflict.

PK conflicts are handled using one of the above conflict handling strategies: server priority, client priority or timestamp priority.

appSync example PHP code

The current appSync example code is available in PHP only. Contributions for other languages and platforms (Java, JS, C# .NET, node.js, Objective C...) are welcome!

The PHP code can be downloaded on appsync.org. It contains following files (version 0.1):

- appsync.php: the core code that implements the synchronization algorithm
- unittest.php: test code that creates a server, 2 clients (client1 and client2), creates objects and executes various synchronization cycles.
- license: license of the source code

The PHP code in appsync.php consists of 3 classes:

- Class Server: implements a server
- Class Client: implements a client
- Class Object: implements one type of objects (with example properties “name” and “value”)

Both server and client have an array “objects”, which simulates the local storage of the server and the clients. The array contains a set of objects of class Object.

The only communication between the client and servers is through 2 methods on the server that the client calls (server->syncToClient and server->syncFromClient). In reality this will be implemented using e.g. REST API’s.

In the example code, most properties and methods are “public” for easy access when executing unit tests and debugging.

Using the appSync pattern in REST API's

The appSync example code currently does not use REST API's. The communication between client and server is simulated: the "server" object is a global variable, and the client calls methods from the server object directly (server->syncToClient and server->syncFromClient). In reality this will typically be implemented using REST API's on the server, called by the client (app).

Note for the following examples: the object property "counter_lastupdate" is NOT exchanged between the clients and server, this is a local property, used by the client and the server to select the data to sync, but this parameter is not part of the JSON !

Using the appSync pattern in REST API's - example

Sync objects of type "booking" from server to client (download):

GET http://yourdomain.com/rest/booking/list/12345

12345 = client.servercounter_lastsync (counter value on server for start of sync)

JSON returned from server to client:

```
{
   objects:
   [
      {guid: FC385707-7027..., pk: 2014-05-10, name: apples, value: 9,
       timestamp_created: 2014-03-20 13:15:00 CET, timestamp_lastupdate: 2014-03-20 13:15:00 CET},
      {guid: AB384887-3563..., pk: 2014-0-12, name: oranges, value: 3,
       timestamp_created: 2014-03-20 12:18:08 CET, timestamp_lastupdate: 2014-03-20 15:17:23 CET},
      {guid: EA585721-9426..., pk: 2014-07-18, name: cherries, value: 6,
       timestamp_created: 2014-03-20 13:16:32 CET, timestamp_lastupdate: 2014-03-20 15:17:58 CET}
   ],
   result:
   {
      statuscode: OK,
      servercounter: 453
   }
}
```

Using the appSync pattern in REST API's - example

Sync objects of type "booking" from client to server (upload):

POST http://yourdomain.com/rest/booking/update

JSON in POST from client to server:

```
{
   objects:
   [
      {guid: FC385707-7027..., pk: 2014-05-11, name: apples, value: 9,
       timestamp_created: 2014-03-20 13:15:00 CET, timestamp_lastupdate: 2014-03-20 13:15:00 CET},
      {guid: AB384887-3563..., pk: 2014-06-12, name: oranges, value: 7,
       timestamp_created: 2014-03-20 12:18:08 CET, timestamp_lastupdate: 2014-03-20 15:23:25 CET},
     {guid: EA585721-9426..., pk: 2014-07-18, name: cherries, value: 9,
       timestamp_created: 2014-03-20 13:16:32 CET, timestamp_lastupdate: 2014-03-20 15:19:43 CET}
   ]
}
```

JSON returned from server to client:

```
{
   result:
   {
      statuscode: OK,
      servercounter: 456
   }
}
```

Unit testing for your synchronization algorithm

When implementing a sync algorithm, it is of course of crucial importance to test it for all possible scenario's, to make sure all data is synced, no data is synced for no reason and no data is lost (overwritten) unwanted.

The appSync code implements following unit tests:

1. Sync from server to client: new objects and object updates (updates from both client and server)
2. Sync from client to server: new objects and object updates (updates from both client and server)
3. Sync from client A to server to other client B
4. No unneeded syncing: e.g. client syncs update to server, client syncs a second time and receives its own update again, this should not occur
5. Syncing of deleted objects (with soft delete: isdeleted=1)
6. Syncing with conflict handling: object is updated on client and on server and then syncing takes place
7. Syncing with primary key conflict: object with same PK is created both on client and on server and then syncing takes place
8. Syncing with primary key conflict: object with same PK is created on client A and client B and then syncing takes place (2 full cycles): object should now be identical on all clients and on server
9. Full sync, with locally created objects that are not synced yet (meaning full sync of both existing and new objects) from server to client and from client to server

Frequently asked questions

What is the difference between data synchronization, REST API's, webservices, replication, mBaaS, websockets (socket.io), node.js etc. ?

- Data synchronization: exchanging data between clients (e.g. mobile apps on mobile devices) and a server
- REST API's: typically used for communication between a server and clients. The clients (apps) will call the REST API's over HTTP(S) to both sync data from and to the server. This means the clients initiate the communication. The input and output parameters required to implement synchronization (e.g. counter) must be implemented in the REST API calls.
- Webservices: a more general term for "API calls", API calls can be implemented using REST and e.g. JSON as data format, but also using SOAP and e.g. XML as data format.
- Replication: a term typically used in data synchronization between databases, in many cases in one direction (e.g. replication from a master DB to a slave DB)
- mBaaS: mobile backend as a service, a platform which provides backend functionality for mobile apps, e.g. out of the box API's that the apps can call to retrieve/send data. mBaaS platforms should ideally provide synchronization capabilities, e.g. based on the appSync patterns and code.
- Websockets and socket.io: a different way for communication between a backend and mobile apps, not using REST API's over HTTP(s). Also when using websockets, a synchronization pattern must be implemented.
- Node.js: a framework based on javascript and socket.io that allows efficient implementation of web-based applications. Node.js can be used to develop backends for mobile apps as well. The appSync team intends to make a node.js version of its example sync code available in the future. Contributions are welcome!

Frequently asked questions

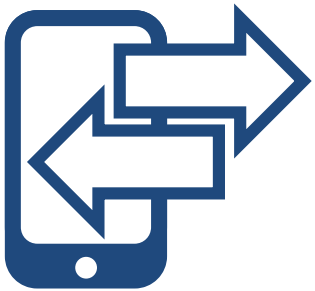
What is the difference between a guid and a PK (primary key) in the appSync example code?

A guid (global unique identifier) looks like this: FC385707-7027-42DF-8118-6F25AC46431B. It is automatically generated in code for each object that is newly created, and is guaranteed to be unique, even across devices. When the client and server sync objects, an object is considered new when the guid is not known yet on the device, and the object is updated if the guid is already known.

A PK is a property or a combination of properties, chosen by the developer, that should be unique for the application to work correctly: for example in our booking app example the PK for objects of type “booking” is date+room number.

What is the difference between the object name and value in the appSync example code ?

Both properties “name” and “value” are just examples, and should be replaced by real object properties in your own code. In the example code, “name” is set on object creation, and never updated during sync, which means it can be used for debugging to track an object as it syncs across devices. “Value” on the other hand is updated in the code from time to time and then synced again.



Visit www.appsync.org to download the source code.