Parse Notes

**Saving Data:**

You don’t have to configure a new class called Gamescore before running the code. Your parse app lazily creates this class for you when it encounters it. PFObject \*object.

Conveniently provided:

objectId, createdAt, and updatedAt -> Represent the time that each object was created and last modified.

You can use saveInBackgroundWithBlock or saveInBackgroundWithTarget:selector: methods to provide additional logic which will run after the save completes.

**Retrieving data**

If you have the objectId you can retrieve the whole PFObject using PFQuery. This is an asynchronous method, with variations for using either blocks or callback methods.

PFQuery \*query = [PFQuery queryWithClassName: @”GameScore”];

[query getObjectInBackgroundWithID:@”xWMyZ4YEGZ” block:^(PFObject \*gameScore, NSError \*error)

{

//do something with the returned PFObject in the gameScore variable.

NSLog(@”%@”, gameScore);

}];

//The inBackground methods are asynchronous, so any code after this will run immediately. Any code that depends on the query result should be moved inside the completion block above.

To get value out of PFObject, you can use objectForKey: method or the [].

int score = [[gameScore objectForKey: @”score”] intValue];

NSString \*playerName = gameScore [@”playerName”];

BOOL cheatMode = [gameScore[@”cheatMode”] boolValue];

The three special value are provided as properties:

NSString \*objectId = gameScore.objectId;

NSDate \*updatedAt = gameScore.updatedAt;

NSDate \*createdAt = gamesScore. createdAt;

If you need to refresh an object you already have with the latest data in the Parse cloud, you can call the refresh method like so:

[myObject refresh];

**The Local Datastore**

Parse lets you store locally. You can use this for data that doesn’t need to be stored in cloud, but this is especially useful for temp storing data so that it can be synced later. To enable the datastore, add libsqlite3.dylib and call [Parse enableLocalDatastore] in your AppDelegate application: didFinishLaunchWithOptions: before calling [Parse setApplicationId:clientKey:].

PFObject \*gameScore = [PFObject objectWithClassName:@”GameScore”];

gameScore[@”score”] = 1337;

gameScore[@”playerName”] = @”Sean Plott”;

gameScore[@”cheatMode”] = @NO;

[gameScore pinInBackground];

As with saving, this recursively stores every object and file that gameScore points to, if it has been fetched from the cloud. Whenever you save changes to the object, or fetch new changes from Parse, the copy in the datastore will automatically be updated, no worrying.

**Retrieve Objects from the Local Datastore**

After you store it, now you gotta get it. To get the data of a specific object you can use PFQuery just like you would while on the network, but using the fromLocalDatastore method to tell it where to get the data.

PFQuery \*query = [PFQuery queryWithClassName:@”GameScore”];

[query froLocalDatastore];

[[query getObjectInBackgroundWithId:@”xWMyZ4YEGZ”] continueWithBlock:^id(BFTask \*task)

{

if (task.error)

{

//something went wrong

return task;

}

return task;

}];

If you already have an instance of the object, you can use the fetchFromLocalDatastoreInBackgroung method.

PFObject \*object = [PFObject objectWithoutDataWithClassName: @”GameScore” objectId:@”xWMyZ4EGZ”];

[[object fetchFromLocalDatastoreInBackground] continueWithBlock:^id(BFTask \*task)

{

if (task.error)

{

//something went wrong

return task;

}

return task;

}];

**Unpinning Objects**

If you want to remove the object from the device and no longer need it you can release it with unpinInBackground.

[gameScore unpinInBackground];

**Saving Objects Offline**

Most save functions execute immediately, and inform your app when the save is complete. If you don’t need to know when the save has finished, you can use saveEventually instead. Advantage is that if the user has no network connection, saveEventually will store the update on the device until a network connection is re-established. If your app is closed before the connection is back, Parse will try again the next time the app is opened. All calls to saveEventually (and deleteEventually) are executed in the order they are called, so it is safe to call saveEventually on an object multiple times.

//create the object

PFObject \*gameScore = [PFObject objectWithClassName: @”GameScore”];

gameScore[@”score”] = @1337;

gameScore[@”playerName”] = @”Sean Plott”];

gameScore[@”cheatMode”] = @NO;

[gameScore saveEventually];

**Updating Objects**

It’s simple. Just set some new data on it and call one of the save methods. Assuming you have saved the object and have the objectId, you can retrieve the PFObject using a PFQuery and update its data:

PFQuery \*query = [PFQuery queryWithClassName: @”GameScore”];

//retrieve the object by id

[query getObjectInBackgroundWithId: @”xWMyZ4YEGZ” block: ^(PFObject \*gameScore, NSError \*error)

{

//now lets update it with some new data. In this case, only cheatMode and score will get sent //to the cloud. playerName hasn’t change.

gamScore[@”cheatMode”] = @YES;

gameScore[@”score”] = @1338;

[gameScore saveInBackground];

}];

The client automatically figures out which data has changed so only “dirty” fields will be sent to Parse. You don’t need to worry about squashing data that you didn’t intend to update.

**Counters**

The above example contains a common use case. The “score” field is a counter that we’ll need to continually update with the players latest score. Using the above method works but it’s cumbersome and can lead to problems if you have multiple clients trying to update at the same counter. To help with storing counter-type data, Parse provides methods that atomically increment (or decrement) any number field. So, the same update can be rewritten as:

[gameScore incrementKey:@”score”];

[gameScore saveInBackground];

You can also increment by any amount using incrementKey:byAmount:.

**Arrays**

There are 3 operators that can be used to atomically change an array field:

* addObject:forKey: and addObjectsFromArray:forKey: append the given objects to the end of the array field.
* addUniqueObjectForKey: and addUniqueObjectsFromArray:forKey add only the given objects which aren’t already contained in an array field to that field. The position of the insert is not guaranteed.
* removeObject:forKey and removeObjectsInArray:forKey: remove all instances of each given object from an array field.

For example, we can add items to the set-like “skills field like so:

[gameScore addUniqueObjectsFromArray: @[@”flying, @”kungfu”] forKey:@”skills”];

[gameScore saveInBackground];

Note that it is not currently possible to atomically add and remove items from an array in the same save. You will have to call save in between every different kind of array operation.

**Deleting Objects**

[gameScore deleteObjectInBackground];

If you want to run a callback when the delete is confirmed, you can use the deleteInBackgroundWithBlock: or delteInBackgroundWithTarget:selector: methods. If you want to block the calling thread, you can use the delete method.

You can delete a single field from an object with the removeObjectForKey method.

//After this, the playerName field will be empty

[gameScore removeObjectForKey: @”playerName”];

//Saves the field deletion to the parse cloud

[gameScore saveInBackground];

**Relational Data**

Objects can have relationship with other objects. To model this behavior, any PFObject can be used as a value in other PFObjects. Internally, the Parse framework will store the referred-to object in just one place, to maintain consistency.

For example, each Comment in a blogging app might correspond to one Post. To create the new Post with a single Comment, you could write:

//Create the post

PFObject \*myPost = [PFObject objectWithClassName:@”Post”];

myPost[@”title”] = @”I’m Hungry”;

myPost[@”content”] = @”Where should we go for lunch?”];

//Create the comment

PFObject \*myComment = [PFObject objectWithClassName:@”Comment”];

myComment[@”content”] = @”Lets do Sushiritto.”;

//Add a relation between the Post and Comment

myComment[@”parent”] = myPost;

//this will save both myPost and myComment

[myComment saveInBackground];

You can also link objects using just their objectsId’s like so:

//Add a relation between the Post with objectId “1zEcyE1Z80” and the comment

myComment[@”parent”] = [PFObject objectWithoutDataWithClassName:@”Post” objectId:@”1zEcyE1Z80”];

By default, when fetching an object, related PFObejcts are not fetched. These objects’ values cannot e retrieved until they have been fetched like so:

PFObject \*post = fetchedComment[@”parent”];

[post fetchIfNeededInBackgroundWithBlock:^(PFObject \*post, NSError \*error)

{

NSString \*title = post[@”title”];

//do something with your title variable

}];

You can also model a many-to-many relation using the PFRelation object. This works similar to an NSArray of PFObjects, except that you don’t need to download all the Objects in a relation at once. This allows the PFRelation to scale to many more objects than the NSArray of PFObject approach. For example, a User may have many Posts that they might like. In this case, you can store the set of Posts that a User likes using relationForKey:. In order to add a post to the list, the code would look something like this:

PFUser \*user = [PFUser currentUser];

PFRelation \*relation = [User relationForKey:@”likes”];

[relation addObject:post];

[user saveInBackground];

You can remove a post from the PFRelation with something like:

[relation removeObject: post];

By default, the list of objects in this relation is not downloaded. You can get the list of Posts by using calling findObjectsInBackgroundWithBlock: on the PFQuery returned by query. The code would look like:

[relation query] findObjectsInBackgroundWithBlock: ^(NSArray \*objects, NSError \*error)

{

if (error)

{

//there was an error

} else

{

//objects has all the Posts the current user like.

}

}];

If you want only a subset of the Posts you can add extra constraints to the PFQuery returned by query like this:

PFQuery \*query = [relation query];

//Add another query constraints.

For more details on PFQuery please look at the query portion of this guide. A PFRelation behaves similar to an NSArray of PFObject, so any queries you can do on arrays of objects (other than includeKey:) you can do on PFRelation.

**Data Types**

So far we’ve used values with type NSString, NSNumber, and PFObject. Parse also supports NSDate, NSData, and NSNull.

You can nest NSDictionary and NSArray objects to store more structured data within a single PFObject.

Some examples:

NSNumber \*number = @42;

NSString \*string = [NSString stringWithFormat: @”The number is %@, number];

NSDate \*date = [NSDate date];

NSData \*data = [@”foo” dataUsingEncoding:NSUTF&StringEncoding];

NSArray \*array = @[string, number];

NSDictionary \*dictionary = @{@”number” : number, @”string” : string};

NSNull \*null = [NSNull null];

PFObject \*bigObject = [PFObject objectWithClassName:@”BigObject”];

bigObject[@”myNumber”] = number;

bigObject[@”myString”] = string;

bigObject[@”myDate”] = date;

bigObject[@”myData”] = data;

bigObject[@”myArray”] = array;

bigObject[@”myDictionary”] = dictionary;

bigObject[@”myNull”] = null;

[bigObject saveInBackground];

We do not recommend storing large pieces of binary data like images or documents using NSData fields on the PFObject. PFObjects should not exceed 128 kilobytes in size. To store more, we recommend you using PFFile.

**Queries**

We’ve already seen how a PFQuery with getObjectWithId: can retrieve a single PFObject from Parse. There are many other ways to retrieve data with a PFQuery – you can retrieve many objects at once, put conditions on the objects you wish to retrieve, cache queries automatically to avoid writing that code yourself, and more.

The general pattern is to create a PFQuery, put conditions on it, and then retrieve a NSArray of matching PFObjects using either findObjectsInBackgroundWithBlock: or findObjectsInBackgroundWithTarget:selector:. For example to retrieve scores with a particular playerName, use the whereKey:equalTo: method to constrain the value for a key.

PFQuery \*query = [PFQuery queryWithClassName:@”GameScore”];

[query whereKey:@”playerName” equalTo:@”Dan Stemoski”];

[query findObjectsInBackgroundWithblock:^(NSArray \*objects, NSError \*error)

{

if (!error)

{

//The find Succeeded

NSLog(@”successfully retrieved %d scores.”, objects.count);

//do something with the found objects

for (PFObject \*object in objects)

{

NSLog(@”%@”, object.objectId);

}

} else

{

//Log details of the failure

NSLog(@”Error: %@,”, error, [error userInfo]);

}

}];

Both findObjectsInBackgroundWithBlock: and findObjectsInBackgroundWithTarget:selector: work similarly in that they assure the network request is done without blocking, and run the block/callback in the main thread. There is a corresponding findObjects method that blocks the calling thread if you are already in a background thread.

//only use this code if you are running it in the background

// thread, or for testing purposes!

PFQuery \*query = [PFQuery queryWithClassName:@”GameScore”];

[query whereKey:@”playerName” equalTo:@”Dan Stemkoski”];

NSArray \*scoreArray = [query findObjects];

**Specifying Constrainsts with NSPredicate**

To get the most out of PFQuery we recommend using its method listed below to add constraints. However, if you prefer using NSPredicate, a subset of the constraints can be specified by providing an NSPredicate when creating your PFQuery.

NSPredicate \*prediate = [NSPredicate predicateWithFormat:@”playerName =

PFUser \*user = [PFUser currentUser];

PFRelation \*relation = [User relationForKey:@”likes”];

[relation addObject:post];

[user saveInBackground];