**Android Guide**

If you haven't installed the SDK yet, please [head over to the QuickStart guide](https://parse.com/apps/quickstart) to get our SDK up and running in Eclipse.

**Introduction**

The Parse platform provides a complete backend solution for your mobile application. Our goal is to totally eliminate the need for writing server code or maintaining servers.

If you're familiar with web frameworks like Ruby on Rails, we've taken many of the same principles and applied them to our platform. In particular, our SDK is ready to use out of the box with minimal configuration on your part.

**Apps**

On Parse, you create an App for each of your mobile applications. Each App has its own application id and client key that you apply to your SDK install. Your account on Parse can accommodate multiple Apps. This is useful even if you have one application, since you can deploy different versions for test and production.

**Objects**

**The ParseObject**

Storing data on Parse is built around the ParseObject. Each ParseObject contains key-value pairs of JSON-compatible data. This data is schemaless, which means that you don't need to specify ahead of time what keys exist on each ParseObject. You simply set whatever key-value pairs you want, and our backend will store it.

For example, let's say you're tracking high scores for a game. A single ParseObject could contain:

|  |
| --- |
| score: 1337, playerName: "Sean Plott", cheatMode: false |

Keys must be alphanumeric strings. Values can be strings, numbers, booleans, or even arrays and objects - anything that can be JSON-encoded.

Each ParseObject has a class name that you can use to distinguish different sorts of data. For example, we could call the high score object a GameScore. We recommend that you NameYourClassesLikeThis and nameYourKeysLikeThis, just to keep your code looking pretty.

**Saving Objects**

Let's say you want to save the GameScore described above to the Parse Cloud. The interface is similar to a Map, plus the saveInBackground method:

|  |
| --- |
| ParseObject gameScore = new ParseObject("GameScore");  gameScore.put("score", 1337);  gameScore.put("playerName", "Sean Plott");  gameScore.put("cheatMode", false);  gameScore.saveInBackground(); |

After this code runs, you will probably be wondering if anything really happened. To make sure the data was saved, you can look at the Data Browser in your app on Parse. You should see something like this:

|  |
| --- |
| objectId: "xWMyZ4YEGZ", score: 1337, playerName: "Sean Plott", cheatMode: false,  createdAt:"2011-06-10T18:33:42Z", updatedAt:"2011-06-10T18:33:42Z" |

There are two things to note here. You didn't have to configure or set up a new Class called GameScore before running this code. Your Parse app lazily creates this Class for you when it first encounters it.

There are also a few fields you don't need to specify that are provided as a convenience. objectId is a unique identifier for each saved object. createdAt and updatedAt represent the time that each object was created and last modified in the cloud. Each of these fields is filled in by Parse, so they don't exist on a ParseObject until a save operation has completed.

**Retrieving Objects**

Saving data to the cloud is fun, but it's even more fun to get that data out again. If you have the objectId, you can retrieve the whole ParseObject using a ParseQuery:

|  |
| --- |
| ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");  query.getInBackground("xWMyZ4YEGZ", new GetCallback<ParseObject>() {    public void done(ParseObject object, ParseException e) {      if (e == null) {        // object will be your game score      } else {        // something went wrong      }    }  }); |

To get the values out of the ParseObject, there's a getX method for each data type:

|  |
| --- |
| int score = gameScore.getInt("score");  String playerName = gameScore.getString("playerName");  boolean cheatMode = gameScore.getBoolean("cheatMode"); |

If you don't know what type of data you're getting out, you can call get(key), but then you probably have to cast it right away anyways. In most situations you should use the typed accessors like getString.

The three special values have their own accessors:

|  |
| --- |
| String objectId = gameScore.getObjectId();  Date updatedAt = gameScore.getUpdatedAt();  Date createdAt = gameScore.getCreatedAt(); |

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

|  |
| --- |
| myObject.fetchInBackground(new GetCallback<ParseObject>() {    public void done(ParseObject object, ParseException e) {      if (e == null) {        // Success!      } else {        // Failure!      }    }  }); |

The code in the GetCallback will be run on the main thread.

**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. The advantage is that if the user currently doesn't have a 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.

|  |
| --- |
| ParseObject gameScore = new ParseObject("GameScore");  gameScore.put("score", 1337);  gameScore.put("playerName", "Sean Plott");  gameScore.put("cheatMode", false);  gameScore.saveEventually(); |

**Updating Objects**

Updating an object is simple. Just set some new data on it and call one of the save methods. For example:

|  |
| --- |
| // Create the object.  final ParseObject gameScore = new ParseObject("GameScore");  gameScore.put("score", 1337);  gameScore.put("playerName", "Sean Plott");  gameScore.put("cheatMode", false);  gameScore.put("skills", Arrays.asList("pwnage", "flying"));  gameScore.saveInBackground(new SaveCallback() {    public void done(ParseException e) {      // Now let's update it with some new data. In this case, only cheatMode and score      // will get sent to the Parse Cloud. playerName hasn't changed.      gameScore.put("score", 1338);      gameScore.put("cheatMode", true);      gameScore.saveInBackground();    }  }); |

Parse automatically figures out which data has changed so only "dirty" fields will be transmitted during a save. You don't need to worry about squashing data in the cloud 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 player's latest score. Using the above method works but it's cumbersome and can lead to problems if you have multiple clients trying to update 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.increment("score");  gameScore.saveInBackground(); |

You can also increment by any amount using increment(key, amount).

**Arrays**

To help with storing array data, there are three operations that can be used to atomically change an array field:

* add and addAll append the given objects to the end of an array field.
* addUnique and addAllUnique 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.
* removeAll removes all instances of the given objects from an array field.

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

|  |
| --- |
| gameScore.addAllUnique("skills", Arrays.asList("flying", "kungfu"));  gameScore.saveInBackground(); |

**Deleting Objects**

To delete an object from the Parse Cloud:

|  |
| --- |
| myObject.deleteInBackground(); |

If you want to run a callback when the delete is confirmed, you can provide a DeleteCallback to the deleteInBackground method. 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 remove method:

|  |
| --- |
| // After this, the playerName field will be empty  myObject.remove("playerName");    // Saves the field deletion to the Parse Cloud  myObject.saveInBackground(); |

**Relational Data**

Objects can have relationships with other objects. To model this behavior, any ParseObject can be used as a value in other ParseObjects. 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 a new Post with a single Comment, you could write:

|  |
| --- |
| // Create the post  ParseObject myPost = new ParseObject("Post");  myPost.put("title", "I'm Hungry");  myPost.put("content", "Where should we go for lunch?");    // Create the comment  ParseObject myComment = new ParseObject("Comment");  myComment.put("content", "Let's do Sushirrito.");    // Add a relation between the Post and Comment  myComment.put("parent", myPost);    // This will save both myPost and myComment  myComment.saveInBackground(); |

You can also link objects using just their objectIds like so:

|  |
| --- |
| // Add a relation between the Post with objectId "1zEcyElZ80" and the comment  myComment.put("parent", ParseObject.createWithoutData("Post", "1zEcyElZ80")); |

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

|  |
| --- |
| fetchedComment.getParseObject("post")      .fetchIfNeededInBackground(new GetCallback<ParseObject>() {          public void done(ParseObject object, ParseException e) {            String title = post.getString("title");          }      }); |

You can also model a many-to-many relation using the ParseRelation object. This works similar to List, except that you don't need to download all the ParseObjects in a relation at once. This allows ParseRelation to scale to many more objects than the List 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 getRelation. In order to add a post to the list, the code would look something like:

|  |
| --- |
| ParseUser user = ParseUser.getCurrentUser();  ParseRelation<ParseObject> relation = user.getRelation("likes");  relation.add(post);  user.saveInBackground(); |

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

|  |
| --- |
| relation.remove(post); |

By default, the list of objects in this relation are not downloaded. You can get the list of Posts by calling findInBackground on the ParseQuery returned by getQuery. The code would look like:

|  |
| --- |
| relation.getQuery().findInBackground(new FindCallback<ParseObject>() {      void done(List<ParseObject> results, ParseException e) {        if (e != null) {          // There was an error        } else {          // results have all the Posts the current user liked.        }      }  }); |

If you want only a subset of the Posts you can add extra constraints to the ParseQuery returned by getQuery. The code would look something like:

|  |
| --- |
| ParseQuery<ParseObject> query = relation.getQuery();  // Add other query constraints. |

For more details on ParseQuery, please look at the [query portion of this guide](https://parse.com/docs/android_guide#queries). A ParseRelation behaves similar to a List for querying purposes, so any queries you can do on lists of objects (other than include) you can do on ParseRelation.

**Data Types**

So far we've used values with type String, int, bool, and ParseObject. Parse also supports java.util.Date, byte[], and JSONObject.NULL.

You can nest JSONObject and JSONArray objects to store more structured data within a single ParseObject.

Some examples:

|  |
| --- |
| int myNumber = 42;  String myString = "the number is " + myNumber;  Date myDate = new Date();    JSONArray myArray = new JSONArray();  myArray.put(myString);  myArray.put(myNumber);    JSONObject myObject = new JSONObject();  myObject.put("number", myNumber);  myObject.put("string", myString);    byte[] myData = { 4, 8, 16, 32 };    ParseObject bigObject = new ParseObject("BigObject");  bigObject.put("myNumber", myNumber);  bigObject.put("myString", myString);  bigObject.put("myDate", myDate);  bigObject.put("myData", myData);  bigObject.put("myArray", myArray);  bigObject.put("myObject", myObject);  bigObject.put("myNull", JSONObject.NULL);  bigObject.saveInBackground(); |

We do not recommend storing large pieces of binary data like images or documents using byte[] fields on ParseObject. ParseObjectss should not exceed 128 kilobytes in size. To store more, we recommend you use ParseFile. See the [guide section](https://parse.com/docs/android_guide#files) for more details.

For more information about how Parse handles data, check out our documentation on [Data & Security](https://parse.com/docs/data).

**Other Operations**

Parse provides a number of convenient operations on ParseObject to help you manage your data.

**Counters**

Many apps keep counter data -- whether it's to track game scores, gold coins, or even sheep. Parse provides an easy way to atomically update your counter data. For example:

|  |
| --- |
| ParseObject player = new ParseObject("Player");  player.put("goldCoins", 1);  player.saveInBackground();    // Later ..  player.increment("goldCoins");  player.saveInBackground(); |

You can also increment by any amount using increment(key, amount).

**Queries**

**Basic Queries**

In many cases, getInBackground isn't powerful enough to specify which objects you want to retrieve. The ParseQuery offers different ways to retrieve a list of objects rather than just a single object.

The general pattern is to create a ParseQuery, put conditions on it, and then retrieve a List of matching ParseObjects using the findInBackground method with a FindCallback. For example, to retrieve scores with a particular playerName, use the whereEqualTo method to constrain the value for a key:

|  |
| --- |
| ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");  query.whereEqualTo("playerName", "Dan Stemkoski");  query.findInBackground(new FindCallback<ParseObject>() {      public void done(List<ParseObject> scoreList, ParseException e) {          if (e == null) {              Log.d("score", "Retrieved " + scoreList.size() + " scores");          } else {              Log.d("score", "Error: " + e.getMessage());          }      }  }); |

findInBackground works similarly to getInBackground in that it assures the network request is done on a background thread, and runs its callback in the main thread.

**Query Constraints**

There are several ways to put constraints on the objects found by a ParseQuery. You can filter out objects with a particular key-value pair with whereNotEqualTo:

|  |
| --- |
| query.whereNotEqualTo("playerName", "Michael Yabuti"); |

You can give multiple constraints, and objects will only be in the results if they match all of the constraints. In other words, it's like an AND of constraints.

|  |
| --- |
| query.whereNotEqualTo("playerName", "Michael Yabuti");  query.whereGreaterThan("playerAge", 18); |

You can limit the number of results with setLimit. By default, results are limited to 100, but anything from 1 to 1000 is a valid limit:

|  |
| --- |
| query.setLimit(10); // limit to at most 10 results |

If you want exactly one result, a more convenient alternative may be to use getFirst or getFirstBackground instead of using find.

|  |
| --- |
| ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");  query.whereEqualTo("playerEmail", "dstemkoski@example.com");  query.getFirstInBackground(new GetCallback<ParseObject>() {    public void done(ParseObject object, ParseException e) {      if (object == null) {        Log.d("score", "The getFirst request failed.");      } else {        Log.d("score", "Retrieved the object.");      }    }  }); |

You can skip the first results with setSkip. This can be useful for pagination:

|  |
| --- |
| query.setSkip(10); // skip the first 10 results |

For sortable types like numbers and strings, you can control the order in which results are returned:

|  |
| --- |
| // Sorts the results in ascending order by the score field  query.orderByAscending("score");    // Sorts the results in descending order by the score field  query.orderByDescending("score"); |

You can add more sort keys to the query as follows:

|  |
| --- |
| // Sorts the results in ascending order by the score field if the previous sort keys are equal.  query.addAscendingOrder("score");    // Sorts the results in descending order by the score field if the previous sort keys are equal.  query.addDescendingOrder("score"); |

For sortable types, you can also use comparisons in queries:

|  |
| --- |
| // Restricts to wins < 50  query.whereLessThan("wins", 50);    // Restricts to wins <= 50  query.whereLessThanOrEqualTo("wins", 50);    // Restricts to wins > 50  query.whereGreaterThan("wins", 50);    // Restricts to wins >= 50  query.whereGreaterThanOrEqualTo("wins", 50); |

If you want to retrieve objects matching several different values, you can use whereContainedIn, providing a collection of acceptable values. This is often useful to replace multiple queries with a single query. For example, if you want to retrieve scores made by any player in a particular list:

|  |
| --- |
| String[] names = {"Jonathan Walsh", "Dario Wunsch", "Shawn Simon"};  query.whereContainedIn("playerName", Arrays.asList(names)); |

If you want to retrieve objects that do not match any of several values you can use whereKey:notContainedIn:, providing an array of acceptable values. For example, if you want to retrieve scores from players besides those in a list:

|  |
| --- |
| String[] names = {"Jonathan Walsh", "Dario Wunsch", "Shawn Simon"};  query.whereNotContainedIn("playerName", Arrays.asList(names)); |

If you want to retrieve objects that have a particular key set, you can use whereExists. Conversely, if you want to retrieve objects without a particular key set, you can use whereDoesNotExist.

|  |
| --- |
| // Finds objects that have the score set  query.whereExists("score");    // Finds objects that don't have the score set  query.whereDoesNotExist("score"); |

You can use the whereMatchesKeyInQuery method to get objects where a key matches the value of a key in a set of objects resulting from another query. For example, if you have a class containing sports teams and you store a user's hometown in the user class, you can issue one query to find the list of users whose hometown teams have winning records. The query would look like:

|  |
| --- |
| ParseQuery<ParseObject> teamQuery = ParseQuery.getQuery("Team");  teamQuery.whereGreaterThan("winPct", 0.5);  ParseQuery<ParseUser> userQuery = ParseUser.getQuery();  userQuery.whereMatchesKeyInQuery("hometown", "city", teamQuery);  userQuery.findInBackground(new FindCallback<ParseUser>() {    void done(List<ParseUser> results, ParseException e) {      // results has the list of users with a hometown team with a winning record    }  }); |

Conversely, to get objects where a key does not match the value of a key in a set of objects resulting from another query, use whereDoesNotMatchKeyInQuery. For example, to find users whose hometown teams have losing records:

|  |
| --- |
| ParseQuery<ParseUser> losingUserQuery = ParseUser.getQuery();  losingUserQuery.whereDoesNotMatchKeyInQuery("hometown", "city", teamQuery);  losingUserQuery.findInBackground(new FindCallback<ParseUser>() {    void done(List<ParseUser> results, ParseException e) {      // results has the list of users with a hometown team with a losing record    }  }); |

You can restrict the fields returned by calling selectKeys with a collection of keys. To retrieve documents that contain only the score and playerName fields (and also special built-in fields such as objectId, createdAt, and updatedAt):

|  |
| --- |
| ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");  query.selectKeys(Arrays.asList("playerName", "score"));;  List<ParseObject> results = query.find(); |

The remaining fields can be fetched later by calling one of the fetchIfNeeded variants on the returned objects:

|  |
| --- |
| ParseObject object = results.get(0);  object.fetchIfNeededInBackground(new GetCallback<ParseObject>() {    public void done(ParseObject object, ParseException e) {      // all fields of the object will now be available here.    }  }); |

**Queries on Array Values**

If a key contains an array value, you can search for objects where the key's array value contains 2 by:

|  |
| --- |
| // Find objects where the array in arrayKey contains the number 2.  query.whereEqualTo("arrayKey", 2); |

You can also search for objects where the key's array value contains each of the values 2, 3, and 4 with the following:

|  |
| --- |
| // Find objects where the array in arrayKey contains all of the numbers 2, 3, and 4.  ArrayList<Integer> numbers = new ArrayList<Integer>();  numbers.add(2);  numbers.add(3);  numbers.add(4);  query.whereContainsAll("arrayKey", numbers); |

**Queries on String Values**

Use whereStartsWith to restrict to string values that start with a particular string. Similar to a MySQL LIKE operator, this is indexed so it is efficient for large datasets:

|  |
| --- |
| // Finds barbecue sauces that start with "Big Daddy's".  ParseQuery<ParseObject> query = ParseQuery.getQuery("BarbecueSauce");  query.whereStartsWith("name", "Big Daddy's"); |

**Relational Queries**

There are several ways to issue queries for relational data. If you want to retrieve objects where a field matches a particular ParseObject, you can use whereEqualTo just like for other data types. For example, if each Comment has a Post object in its post field, you can fetch comments for a particular Post:

|  |
| --- |
| // Assume ParseObject myPost was previously created.  ParseQuery<ParseObject> query = ParseQuery.getQuery("Comment");  query.whereEqualTo("post", myPost);    query.findInBackground(new FindCallback<ParseObject>() {    public void done(List<ParseObject> commentList, ParseException e) {      // commentList now has the comments for myPost    }  }); |

If you want to retrieve objects where a field contains a ParseObject that matches a different query, you can use whereMatchesQuery. Note that the default limit of 100 and maximum limit of 1000 apply to the inner query as well, so with large data sets you may need to construct queries carefully to get the desired behavior. In order to find comments for posts containing images, you can do:

|  |
| --- |
| ParseQuery<ParseObject> innerQuery = ParseQuery.getQuery("Post");  innerQuery.whereExists("image");  ParseQuery<ParseObject> query = ParseQuery.getQuery("Comment");  query.whereMatchesQuery("post", innerQuery);  query.findInBackground(new FindCallback<ParseObject>() {    public void done(List<ParseObject> commentList, ParseException e) {      // comments now contains the comments for posts with images.    }  }); |

If you want to retrieve objects where a field contains a ParseObject that does not match a different query, you can use whereDoesNotMatchQuery. In order to find comments for posts without images, you can do:

|  |
| --- |
| ParseQuery<ParseObject> innerQuery = ParseQuery.getQuery("Post");  innerQuery.whereExists("image");  ParseQuery<ParseObject> query = ParseQuery.getQuery("Comment");  query.whereDoesNotMatchQuery("post", innerQuery);  query.findInBackground(new FindCallback<ParseObject>() {    public void done(List<ParseObject> commentList, ParseException e) {      // comments now contains the comments for posts without images.    }  }); |

In some situations, you want to return multiple types of related objects in one query. You can do this with the include method. For example, let's say you are retrieving the last ten comments, and you want to retrieve their related posts at the same time:

|  |
| --- |
| ParseQuery<ParseObject> query = ParseQuery.getQuery("Comment");    // Retrieve the most recent ones  query.orderByDescending("createdAt");    // Only retrieve the last ten  query.setLimit(10);    // Include the post data with each comment  query.include("post");    query.findInBackground(new FindCallback<ParseObject>() {    public void done(List<ParseObject> commentList, ParseException e) {      // commentList now contains the last ten comments, and the "post"      // field has been populated. For example:      for (ParseObject comment : commentList) {        // This does not require a network access.        ParseObject post = comment.getParseObject("post");        Log.d("post", "retrieved a related post");      }    }  }); |

You can also do multi level includes using dot notation. If you wanted to include the post for a comment and the post's author as well you can do:

|  |
| --- |
| query.include("post.author"); |

You can issue a query with multiple fields included by calling include multiple times. This functionality also works with ParseQuery helpers like getFirst() and getInBackground().

**Caching Queries**

It's often useful to cache the result of a query on disk. This lets you show data when the user's device is offline, or when the app has just started and network requests have not yet had time to complete. Parse takes care of automatically flushing the cache when it takes up too much space.

The default query behavior doesn't use the cache, but you can enable caching with setCachePolicy. For example, to try the network and then fall back to cached data if the network is not available:

|  |
| --- |
| query.setCachePolicy(ParseQuery.CachePolicy.NETWORK\_ELSE\_CACHE);  query.findInBackground(new FindCallback<ParseObject>() {  public void done(List<ParseObject> scoreList, ParseException e) {    if (e == null) {      // Results were successfully found, looking first on the      // network and then on disk.    } else {      // The network was inaccessible and we have no cached data      // for this query.    }  }); |

Parse provides several different cache policies:

* IGNORE\_CACHE   
  The query does not load from the cache or save results to the cache. IGNORE\_CACHE is the default cache policy.
* CACHE\_ONLY   
  The query only loads from the cache, ignoring the network. If there are no cached results, that causes a ParseException.
* NETWORK\_ONLY   
  The query does not load from the cache, but it will save results to the cache.
* CACHE\_ELSE\_NETWORK   
  The query first tries to load from the cache, but if that fails, it loads results from the network. If neither cache nor network succeed, there is a ParseException.
* NETWORK\_ELSE\_CACHE   
  The query first tries to load from the network, but if that fails, it loads results from the cache. If neither network nor cache succeed, there is a ParseException.
* CACHE\_THEN\_NETWORK   
  The query first loads from the cache, then loads from the network. In this case, the FindCallback will actually be called twice - first with the cached results, then with the network results. This cache policy can only be used asynchronously with findInBackground.

If you need to control the cache's behavior, you can use methods provided in ParseQuery to interact with the cache. You can do the following operations on the cache:

* Check to see if there is a cached result for the query with:

|  |
| --- |
| boolean isInCache = query.hasCachedResult(); |

* Remove any cached results for a query with:

|  |
| --- |
| query.clearCachedResult(); |

* Remove cached results for all queries with:

|  |
| --- |
| ParseQuery.clearAllCachedResults(); |

* Control the maximum age of a cached result with:

|  |
| --- |
| query.setMaxCacheAge(TimeUnit.DAYS.toMillis(1)); |

Query caching also works with ParseQuery helpers including getFirst() and getInBackground().

**Counting Objects**

If you just need to count how many objects match a query, but you do not need to retrieve all the objects that match, you can use count instead of find. For example, to count how many games have been played by a particular player:

|  |
| --- |
| ParseQuery<ParseObject> query = ParseQuery.getQuery("GameScore");  query.whereEqualTo("playerName", "Sean Plott");  query.countInBackground(new CountCallback() {    public void done(int count, ParseException e) {      if (e == null) {        // The count request succeeded. Log the count        Log.d("score", "Sean has played " + count + " games");      } else {        // The request failed      }    }  }); |

If you want to block the calling thread, you can also use the synchronous query.count() method.

For classes with over 1000 objects, count operations are limited by timeouts. They may routinely yield timeout errors or return results that are only approximately correct. Thus, it is preferable to architect your application to avoid this sort of count operation.

**Compound Queries**

If you want to find objects that match one of several queries, you can use ParseQuery.or method to construct a query that is an or of the queries passed in. For instance if you want to find players who either have a lot of wins or a few wins, you can do:

|  |
| --- |
| ParseQuery<ParseObject> lotsOfWins = ParseQuery.getQuery("Player");  lotsOfWins.whereGreaterThan(150);    ParseQuery<ParseObject> fewWins = ParseQuery.getQuery("Player");  fewWins.whereLessThan(5);    List<ParseQuery<ParseObject>> queries = new ArrayList<ParseQuery<ParseObject>>();  queries.add(lotsOfWins);  queries.add(fewWins);    ParseQuery<ParseObject> mainQuery = ParseQuery.or(queries);  mainQuery.findInBackground(new FindCallback<ParseObject>() {    public void done(List<ParseObject> results, ParseException e) {      // results has the list of players that win a lot or haven't won much.    }  }); |

You can add additional constraints to the newly created ParseQuery that act as an 'and' operator.

Note that we do not, however, support non-filtering constraints (e.g. setLimit, skip, orderBy..., include) in the subqueries of the compound query.

**Subclasses**

Parse is designed to get you up and running as quickly as possible. You can access all of your data using the ParseObject class and access any field with get(). In mature codebases, subclasses have many advantages, including terseness, extensibility, and support for autocomplete. Subclassing is completely optional, but can transform this code:

|  |
| --- |
| ParseObject shield = new ParseObject("Armor");  shield.put("displayName", "Wooden Shield");  shield.put("fireproof", false);  shield.put("rupees", 50); |

Into this:

|  |
| --- |
| Armor shield = new Armor();  shield.setDisplayName("Wooden Shield");  shield.setFireproof(false);  shield.setRupees(50); |

**Subclassing ParseObject**

To create a ParseObject subclass:

1. Declare a subclass which extends ParseObject.
2. Add a @ParseClassName annotation. Its value should be the string you would pass into the ParseObject constructor, and makes all future class name references unnecessary.
3. Ensure that your subclass has a public default (i.e. zero-argument) constructor. You must not modify any ParseObject fields in this constructor.
4. Call ParseObject.registerSubclass(YourClass.class) in your Application constructor before calling Parse.initialize().

The following code sucessfully implements and registers the Armor subclass of ParseObject:

|  |
| --- |
| // Armor.java  import com.parse.ParseObject;  import com.parse.ParseClassName;    @ParseClassName("Armor")  public class Armor extends ParseObject {  }    // App.java  import com.parse.Parse;  import android.app.Application;    public class App extends Application {    @Override    public void onCreate() {      super.onCreate();        ParseObject.registerSubclass(Armor.class);      Parse.initialize(this, PARSE\_APPLICATION\_ID, PARSE\_CLIENT\_KEY);    }  } |

**Accessors, Mutators, and Methods**

Adding methods to your ParseObject subclass helps encapsulate logic about the class. You can keep all your logic about a subject in one place rather than using separate classes for business logic and storage/transmission logic.

You can add accessors and mutators for the fields of your ParseObject easily. Declare the getter and setter for the field as you normally would, but implement them in terms of get() and put(). The following example creates a displayName field in the Armor class:

|  |
| --- |
| // Armor.java  @ParseClassName("Armor")  public class Armor extends ParseObject {    public String getDisplayName() {      return getString("displayName");    }    public void setDisplayName(String value) {      put("displayName", value);    }  } |

You can now access the displayName field using armor.getDisplayName() and assign to it using armor.setDisplayName("Wooden Sword"). This allows your IDE to provide autocompletion as you develop your app and allows typos to be caught at compile-time.

Accessors and mutators of various types can be easily defined in this manner using the various forms of get() such as getInt(), getParseFile(), or getMap().

If you need more complicated logic than simple field access, you can declare your own methods as well:

|  |
| --- |
| public void takeDamage(int amount) {    // Decrease the armor's durability and determine whether it has broken    increment("durability", -amount);    if (getDurability() < 0) {      setBroken(true);    }  } |

**Initializing Subclasses**

You should create new instances of your subclasses using the constructors you have defined. Your subclass must define a public default constructor that does not modify fields of the ParseObject, which will be used throughout the Parse SDK to create strongly-typed instances of your subclass.

To create a reference to an existing object, use ParseObject.createWithoutData():

|  |
| --- |
| Armor armorReference = ParseObject.createWithoutData(Armor.class, armor.getObjectId()); |

**Queries**

You can get a query for objects of a particular subclass using the static method ParseObject.getQuery(). The following example queries for armors that the user can afford:

|  |
| --- |
| ParseQuery<Armor> query = ParseObject.getQuery(Armor.class);  query.whereLessThanOrEqualTo("rupees", ParseUser.getCurrentUser().get("rupees"));  query.findInBackground(new FindCallback<Armor>() {    @Override    public void done(List<Armor> results, ParseException e) {      for (Armor a : results) {        // ...      }    }  }); |

**Files**

**The ParseFile**

ParseFile lets you store application files in the cloud that would otherwise be too large or cumbersome to fit into a regular ParseObject. The most common use case is storing images but you can also use it for documents, videos, music, and any other binary data (up to 10 megabytes).

Getting started with ParseFile is easy. First, you'll need to have the data in byte[] form and then create a ParseFile with it. In this example, we'll just use a string:

|  |
| --- |
| byte[] data = "Working at Parse is great!".getBytes();  ParseFile file = new ParseFile("resume.txt", data); |

Notice in this example that we give the file a name of resume.txt. There's two things to note here:

* You don't need to worry about filename collisions. Each upload gets a unique identifier so there's no problem with uploading multiple files named resume.txt.
* It's important that you give a name to the file that has a file extension. This lets Parse figure out the file type and handle it accordingly. So, if you're storing PNG images, make sure your filename ends with .png.

Next you'll want to save the file up to the cloud. As with ParseObject, there are many variants of the save method you can use depending on what sort of callback and error handling suits you.

|  |
| --- |
| file.saveInBackground(); |

Finally, after the save completes, you can associate a ParseFile onto a ParseObject just like any other piece of data:

|  |
| --- |
| ParseObject jobApplication = new ParseObject("JobApplication");  jobApplication.put("applicantName", "Joe Smith");  jobApplication.put("applicantResumeFile", file);  jobApplication.saveInBackground(); |

Retrieving it back involves calling one of the getData variants on the ParseObject. Here we retrieve the resume file off another JobApplication object:

|  |
| --- |
| ParseFile applicantResume = (ParseFile)anotherApplication.get("applicantResumeFile");  applicantResume.getDataInBackground(new GetDataCallback() {    public void done(byte[] data, ParseException e) {      if (e == null) {        // data has the bytes for the resume      } else {        // something went wrong      }    }  }); |

Just like on ParseObject, you will most likely want to use the background version of getData.

**Progress**

It's easy to get the progress of both uploads and downloads using ParseFile by passing a ProgressCallback to saveInBackground and getDataInBackground. For example:

|  |
| --- |
| byte[] data = "Working at Parse is great!".getBytes();  ParseFile file = new ParseFile("resume.txt", data);    file.saveInBackground(new SaveCallback() {    public void done(ParseException e) {      // Handle success or failure here ...    }  }, new ProgressCallback() {    public void done(Integer percentDone) {      // Update your progress spinner here. percentDone will be between 0 and 100.    }  }); |

**Analytics**

Our initial analytics hook allows you to track your application being launched. By adding the following line to the onCreate method of your main Activity, you'll begin to collect data on when and how often your application is opened.

|  |
| --- |
| ParseAnalytics.trackAppOpened(getIntent()); |

Graphs and breakdowns of your statistics are accessible from your app's Dashboard.

Further analytics are available around push notification delivery and open rates. Take a look at the [Tracking Pushes and App Opens subsection](https://parse.com/docs/push_guide#receiving-tracking/Android) of our Push Guide for more detailed information on handling notification payloads and push-related callbacks.

**Push Notifications**

To learn more about push check out our [Push Notification Guide](https://parse.com/docs/push_guide#top/Android)!

**Users**

At the core of many apps, there is a notion of user accounts that lets users access their information in a secure manner. We provide a specialized user class called ParseUser that automatically handles much of the functionality required for user account management.

With this class, you'll be able to add user account functionality in your app.

ParseUser is a subclass of the ParseObject, and has all the same features, such as flexible schema, automatic persistence, and a key value interface. All the methods that are on ParseObject also exist in ParseUser. The difference is that ParseUser has some special additions specific to user accounts.

**Properties**

ParseUser has several properties that set it apart from ParseObject:

* username: The username for the user (required).
* password: The password for the user (required on signup).
* email: The email address for the user (optional).

We'll go through each of these in detail as we run through the various use cases for users. Keep in mind that if you set username and email using the setters, you do not need to set it using the put method.

**Signing Up**

The first thing your app will do is probably ask the user to sign up. The following code illustrates a typical sign up:

|  |
| --- |
| ParseUser user = new ParseUser();  user.setUsername("my name");  user.setPassword("my pass");  user.setEmail("email@example.com");    // other fields can be set just like with ParseObject  user.put("phone", "650-253-0000");    user.signUpInBackground(new SignUpCallback() {    public void done(ParseException e) {      if (e == null) {        // Hooray! Let them use the app now.      } else {        // Sign up didn't succeed. Look at the ParseException        // to figure out what went wrong      }    }  }); |

This call will asynchronously create a new user in your Parse App. Before it does this, it checks to make sure that both the username and email are unique. Also, it securely hashes the password in the cloud. We never store passwords in plaintext, nor will we ever transmit passwords back to the client in plaintext.

Note that we used the signUpInBackground method, not the saveInBackground method. New ParseUsers should always be created using the signUpInBackground (or signUp) method. Subsequent updates to a user can be done by calling save.

The signUpInBackground method comes in various flavors, with the ability to pass back errors, and also synchronous versions. As usual, we highly recommend using the asynchronous versions when possible, so as not to block the UI in your app. You can read more about these specific methods in our [API docs](https://parse.com/docs/android/).

If a signup isn't successful, you should read the error object that is returned. The most likely case is that the username or email has already been taken by another user. You should clearly communicate this to your users, and ask them try a different username.

You are free to use an email addresses as the username. Simply ask your users to enter their email, but fill it in the username property — ParseUser will work as normal. We'll go over how this is handled in the reset password section.

**Logging In**

Of course, after you allow users to sign up, you need be able to let them log in to their account in the future. To do this, you can use the class method logInInBackground.

|  |
| --- |
| ParseUser.logInInBackground("Jerry", "showmethemoney", new LogInCallback() {    public void done(ParseUser user, ParseException e) {      if (user != null) {        // Hooray! The user is logged in.      } else {        // Signup failed. Look at the ParseException to see what happened.      }    }  }); |

**Verifying Emails**

Enabling email verification in an application's settings allows the application to reserve part of its experience for users with confirmed email addresses. Email verification adds the emailVerified key to the ParseUser object. When a ParseUser's email is set or modified, emailVerified is set to false. Parse then emails the user a link which will set emailVerified to true.

There are three emailVerified states to consider:

1. true - the user confirmed his or her email address by clicking on the link Parse emailed them. ParseUsers can never have a true value when the user account is first created.
2. false - at the time the ParseUser object was last fetched, the user had not confirmed his or her email address. If emailVerified is false, consider calling fetch() on the ParseUser.
3. *missing* - the ParseUser was created when email verification was off or the ParseUser does not have an email.

**Current User**

It would be bothersome if the user had to log in every time they open your app. You can avoid this by using the cached currentUser object.

Whenever you use any signup or login methods, the user is cached on disk. You can treat this cache as a session, and automatically assume the user is logged in:

|  |
| --- |
| ParseUser currentUser = ParseUser.getCurrentUser();  if (currentUser != null) {    // do stuff with the user  } else {    // show the signup or login screen  } |

You can clear the current user by logging them out:

|  |
| --- |
| ParseUser.logOut();  ParseUser currentUser = ParseUser.getCurrentUser(); // this will now be null |

**Anonymous Users**

Being able to associate data and objects with individual users is highly valuable, but sometimes you want to be able to do this without forcing a user to specify a username and password.

An anonymous user is a user that can be created without a username and password but still has all of the same capabilities as any other ParseUser. After logging out, an anonymous user is abandoned, and its data is no longer accessible.

You can create an anonymous user using ParseAnonymousUtils:

|  |
| --- |
| ParseAnonymousUtils.logIn(new LogInCallback() {    @Override    public void done(ParseUser user, ParseException e) {      if (e != null) {        Log.d("MyApp", "Anonymous login failed.");      } else {        Log.d("MyApp", "Anonymous user logged in.");      }    }  }); |

You can convert an anonymous user into a regular user by setting the username and password, then calling signUp(), or by logging in or linking with a service like [Facebook](https://parse.com/docs/android_guide#fbusers) or [Twitter](https://parse.com/docs/android_guide#twitterusers). The converted user will retain all of its data. To determine whether the current user is an anonymous user, you can check ParseAnonymousUtils.isLinked():

|  |
| --- |
| if (ParseAnonymousUtils.isLinked(ParseUser.getCurrentUser())) {    enableSignUpButton();  } else {    enableLogOutButton();  } |

Anonymous users can also be automatically created for you without requiring a network request, so that you can begin working with your user immediately when your application starts. When you enable automatic anonymous user creation at application startup, ParseUser.getCurrentUser() will never be null. The user will automatically be created in the cloud the first time the user or any object with a relation to the user is saved. Until that point, the user's object ID will be null. Enabling automatic user creation makes associating data with your users painless. For example, in your Application.onCreate() method, you might write:

|  |
| --- |
| ParseUser.enableAutomaticUser();  ParseUser.getCurrentUser().increment("RunCount");  ParseUser.getCurrentUser().saveInBackground(); |

**Security For User Objects**

The ParseUser class is secured by default. Data stored in a ParseUser can only be modified by that user. By default, the data can still be read by any client. Thus, some ParseUser objects are authenticated and can be modified, whereas others are read-only.

Specifically, you are not able to invoke any of the save or delete type methods unless the ParseUser was obtained using an authenticated method, like logIn or signUp. This ensures that only the user can alter their own data.

The following illustrates this security policy:

|  |
| --- |
| ParseUser user = ParseUser.logIn("my\_username", "my\_password");  user.setUsername("my\_new\_username"); // attempt to change username  user.saveInBackground(); // This succeeds, since the user was authenticated on the device    // Get the user from a non-authenticated manner  ParseQuery<ParseUser> query = ParseUser.getQuery();  query.getInBackground(user.getObjectId(), new GetCallback<ParseUser>() {    public void done(ParseUser object, ParseException e) {      object.setUsername("another\_username");        // This will throw an exception, since the ParseUser is not authenticated      object.saveInBackground();    }  }); |

The ParseUser obtained from getCurrentUser() will always be authenticated.

If you need to check if a ParseUser is authenticated, you can invoke the isAuthenticated() method. You do not need to check isAuthenticated() with ParseUser objects that are obtained via an authenticated method.

**Security for Other Objects**

The same security model that applies to the ParseUser can be applied to other objects. For any object, you can specify which users are allowed to read the object, and which users are allowed to modify an object. To support this type of security, each object has an [access control list](http://en.wikipedia.org/wiki/Access_control_list), implemented by the ParseACL class.

The simplest way to use a ParseACL is to specify that an object may only be read or written by a single user. To create such an object, there must first be a logged in ParseUser. Then, new ParseACL(user) generates a ParseACL that limits access to that user. An object's ACL is updated when the object is saved, like any other property. Thus, to create a private note that can only be accessed by the current user:

|  |
| --- |
| ParseObject privateNote = new ParseObject("Note");  privateNote.put("content", "This note is private!");  privateNote.setACL(new ParseACL(ParseUser.getCurrentUser()));  privateNote.saveInBackground(); |

This note will then only be accessible to the current user, although it will be accessible to any device where that user is signed in. This functionality is useful for applications where you want to enable access to user data across multiple devices, like a personal todo list.

Permissions can also be granted on a per-user basis. You can add permissions individually to a ParseACL using setReadAccess and setWriteAccess. For example, let's say you have a message that will be sent to a group of several users, where each of them have the rights to read and delete that message:

|  |
| --- |
| ParseObject groupMessage = new ParseObject("Message");  ParseACL groupACL = new ParseACL();    // userList is an Iterable<ParseUser> with the users we are sending this message to.  for (ParseUser user : userList) {    groupACL.setReadAccess(user, true);    groupACL.setWriteAccess(user, true);  }    groupMessage.setACL(groupACL);  groupMessage.saveInBackground(); |

You can also grant permissions to all users at once using setPublicReadAccess and setPublicWriteAccess. This allows patterns like posting comments on a message board. For example, to create a post that can only be edited by its author, but can be read by anyone:

|  |
| --- |
| ParseObject publicPost = new ParseObject("Post");  ParseACL postACL = new ParseACL(ParseUser.getCurrentUser());  postACL.setPublicReadAccess(true);  publicPost.setACL(postACL);  publicPost.saveInBackground(); |

To help ensure that your users' data is secure by default, you can set a default ACL to be applied to all newly-created ParseObjects:

|  |
| --- |
| ParseACL.setDefaultACL(defaultACL, true); |

In the code above, the second parameter to setDefaultACL tells Parse to ensure that the default ACL assigned at the time of object creation allows read and write access to the current user at that time. Without this setting, you would need to reset the defaultACL every time a user logs in or out so that the current user would be granted access appropriately. With this setting, you can ignore changes to the current user until you explicitly need to grant different kinds of access.

Default ACLs make it easy to create apps that follow common access patterns. An application like Twitter, for example, where user content is generally visible to the world, might set a default ACL such as:

|  |
| --- |
| ParseACL defaultACL = new ParseACL();  defaultACL.setPublicReadAccess(true);  ParseACL.setDefaultACL(defaultACL, true); |

For an application like Dropbox, where a user's data is only accessible by the user itself unless explicit permission is given, you would provide a default ACL where only the current user is given access:

|  |
| --- |
| ParseACL.setDefaultACL(new ParseACL(), true); |

An application that logs data to Parse but doesn't provide any user access to that data would instead deny access to the current user while providing a restrictive ACL:

|  |
| --- |
| ParseACL.setDefaultACL(new ParseACL(), false); |

Operations that are forbidden, such as deleting an object that you do not have write access to, result in a ParseException.OBJECT\_NOT\_FOUND error code. For security purposes, this prevents clients from distinguishing which object ids exist but are secured, versus which object ids do not exist at all.

**Resetting Passwords**

It's a fact that as soon as you introduce passwords into a system, users will forget them. In such cases, our library provides a way to let them securely reset their password.

To kick off the password reset flow, ask the user for their email address, and call:

|  |
| --- |
| ParseUser.requestPasswordResetInBackground("myemail@example.com",                                             new RequestPasswordResetCallback() {    public void done(ParseException e) {      if (e == null) {        // An email was successfully sent with reset instructions.      } else {        // Something went wrong. Look at the ParseException to see what's up.      }    }  }); |

This will attempt to match the given email with the user's email or username field, and will send them a password reset email. By doing this, you can opt to have users use their email as their username, or you can collect it separately and store it in the email field.

The flow for password reset is as follows:

1. User requests that their password be reset by typing in their email.
2. Parse sends an email to their address, with a special password reset link.
3. User clicks on the reset link, and is directed to a special Parse page that will allow them type in a new password.
4. User types in a new password. Their password has now been reset to a value they specify.

Note that the messaging in this flow will reference your app by the name that you specified when you created this app on Parse.

**Querying**

To query for users, you need to use the special user query:

|  |
| --- |
| ParseQuery<ParseUser> query = ParseUser.getQuery();  query.whereEqualTo("gender", "female");  query.findInBackground(new FindCallback<ParseUser>() {    public void done(List<ParseUser> objects, ParseException e) {      if (e == null) {          // The query was successful.      } else {          // Something went wrong.      }    }  }); |

In addition, you can use get to get a ParseUser by id.

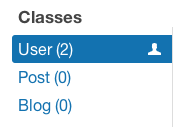
**Associations**

Associations involving a ParseUser work right of the box. For example, let's say you're making a blogging app. To store a new post for a user and retrieve all their posts:

|  |
| --- |
| ParseUser user = ParseUser.getCurrentUser();    // Make a new post  ParseObject post = new ParseObject("Post");  post.put("title", "My New Post");  post.put("body", "This is some great content.");  post.put("user", user);  post.saveInBackground();    // Find all posts by the current user  ParseQuery<ParseObject> query = ParseQuery.getQuery("Post");  query.whereEqualTo("user", user);  query.findInBackground(new FindCallback<ParseObject>() { ... }); |

**Users in the Data Browser**

The User class is a special class that is dedicated to storing ParseUser objects. In the data browser, you'll see a little person icon next to the User class:



**Roles**

As your app grows in scope and user-base, you may find yourself needing more coarse-grained control over access to pieces of your data than user-linked ACLs can provide. To address this requirement, Parse supports a form of [Role-based Access Control](http://en.wikipedia.org/wiki/Role-based_access_control). Roles provide a logical way of grouping users with common access privileges to your Parse data. Roles are named objects that contain users and other roles. Any permission granted to a role is implicitly granted to its users as well as to the users of any roles that it contains.

For example, in your application with curated content, you may have a number of users that are considered "Moderators" and can modify and delete content created by other users. You may also have a set of users that are "Administrators" and are allowed all of the same privileges as Moderators, but can also modify the global settings for the application. By adding users to these roles, you can ensure that new users can be made moderators or administrators, without having to manually grant permission to every resource for each user.

We provide a specialized class called ParseRole that represents these role objects in your client code. ParseRole is a subclass of ParseObject, and has all of the same features, such as a flexible schema, automatic persistence, and a key value interface. All the methods that are on ParseObject also exist on ParseRole. The difference is that ParseRole has some additions specific to management of roles.

**Properties**

ParseRole has several properties that set it apart from ParseObject:

* name: The name for the role. This value is required, and can only be set once as a role is being created. The name must consist of alphanumeric characters, spaces, -, or \_. This name will be used to identify the Role without needing its objectId.
* users: A [relation](https://parse.com/docs/android_guide#objects-pointers) to the set of users that will inherit permissions granted to the containing role.
* roles: A [relation](https://parse.com/docs/android_guide#objects-pointers) to the set of roles whose users and roles will inherit permissions granted to the containing role.

**Security for Role Objects**

The ParseRole uses the same security scheme (ACLs) as all other objects on Parse, except that it requires an ACL to be set explicitly. Generally, only users with greatly elevated privileges (e.g. a master user or Administrator) should be able to create or modify a Role, so you should define its ACLs accordingly. Remember, if you give write-access to a ParseRole to a user, that user can add other users to the role, or even delete the role altogether.

To create a new ParseRole, you would write:

|  |
| --- |
| // By specifying no write privileges for the ACL, we can ensure the role cannot be altered.  ParseACL roleACL = new ParseACL();  roleACL.setPublicReadAccess(true);  ParseRole role = new ParseRole("Administrator", roleACL);  role.saveInBackground(); |

You can add users and roles that should inherit your new role's permissions through the "users" and "roles" relations on ParseRole:

|  |
| --- |
| ParseRole role = new ParseRole(roleName, roleACL);  for (ParseUser user : usersToAddToRole) {    role.getUsers().add(user)  }  for (ParseRole childRole : rolesToAddToRole) {    role.getRoles().add(childRole);  }  role.saveInBackground(); |

Take great care when assigning ACLs to your roles so that they can only be modified by those who should have permissions to modify them.

**Security for Other Objects**

Now that you have created a set of roles for use in your application, you can use them with ACLs to define the privileges that their users will receive. Each ParseObject can specify a ParseACL, which provides an access control list that indicates which users and roles should be granted read or write access to the object.

Giving a role read or write permission to an object is straightforward. You can either use the ParseRole:

|  |
| --- |
| ParseRole moderators = /\* Query for some ParseRole \*/;  ParseObject wallPost = new ParseObject("WallPost");  ParseACL postACL = new ParseACL();  postACL.setRoleWriteAccess(moderators);  wallPost.setACL(postACL);  wallPost.saveInBackground(); |

You can avoid querying for a role by specifying its name for the ACL:

|  |
| --- |
| ParseObject wallPost = new ParseObject("WallPost");  ParseACL postACL = new ParseACL();  postACL.setRoleWriteAccess("Moderators", true);  wallPost.setACL(postACL);  wallPost.save(); |

Role-based ParseACLs can also be used when specifying default ACLs for your application, making it easy to protect your users' data while granting access to users with additional privileges. For example, a moderated forum application might specify a default ACL like this:

|  |
| --- |
| ParseACL defaultACL = new ParseACL();  // Everybody can read objects created by this user  defaultACL.setPublicReadAccess(true);  // Moderators can also modify these objects  defaultACL.setRoleWriteAccess("Moderators");  // And the user can read and modify its own objects  ParseACL.setDefaultACL(defaultACL, true); |

**Role Hierarchy**

As described above, one role can contain another, establishing a parent-child relationship between the two roles. The consequence of this relationship is that any permission granted to the parent role is implicitly granted to all of its child roles.

These types of relationships are commonly found in applications with user-managed content, such as forums. Some small subset of users are "Administrators", with the highest level of access to tweaking the application's settings, creating new forums, setting global messages, and so on. Another set of users are "Moderators", who are responsible for ensuring that the content created by users remains appropriate. Any user with Administrator privileges should also be granted the permissions of any Moderator. To establish this relationship, you would make your "Administrators" role a child role of "Moderators", like this:

|  |
| --- |
| ParseRole administrators = /\* Your "Administrators" role \*/;  ParseRole moderators = /\* Your "Moderators" role \*/;  moderators.getRoles().add(administrators);  moderators.saveInBackground(); |

**Facebook Users**

Parse provides an easy way to integrate Facebook with your application. The Facebook SDK can be used with our SDK, and is integrated with the ParseUser class to make linking your users to their Facebook identities easy.

Using our Facebook integration, you can associate an authenticated Facebook user with a ParseUser. With just a few lines of code, you'll be able to provide a "log in with Facebook" option in your app, and be able to save their data to Parse.

**Setup**

To start using Facebook with Parse, you need to:

1. [Set up a Facebook app](https://developers.facebook.com/apps), if you haven't already.
2. Add your application's Facebook Application ID on your Parse application's settings page.
3. Follow Facebook's instructions for [getting started with the Facebook SDK](https://developers.facebook.com/docs/getting-started/facebook-sdk-for-android/3.0/) to create an app linked to the Facebook SDK. Once you get to Step 6, stop after linking the Facebook SDK project and configuring the Facebook app ID. You can use our guide to attach your Parse users to their Facebook accounts when logging in.
4. Add the following where you initialize the Parse SDK in your Application.onCreate()

|  |
| --- |
| ParseFacebookUtils.initialize("YOUR FACEBOOK APP ID"); |

Facebook's Android SDK provides an enhanced login experience on devices that have [Facebook's official Android app](https://market.android.com/details?id=com.facebook.katana) installed. This allows users of apps that support Facebook login to sign in directly through the Facebook app, using credentials that are already on the device. If the Facebook app is not installed, the default dialog-based authentication will be used. Facebook calls this feature "Single sign-on," and requires you to override onActivityResult() in your calling Activity to invoke finishAuthentication().

|  |
| --- |
| @Override  protected void onActivityResult(int requestCode, int resultCode, Intent data) {    super.onActivityResult(requestCode, resultCode, data);    ParseFacebookUtils.finishAuthentication(requestCode, resultCode, data);  } |

If your Activity is already using onActivityResult(), you can avoid requestCode collisions by calling the versions of link() and logIn() that take an activityCode parameter and specifying a code you know to be unique. Otherwise, a sensible default activityCode will be used.

If you encounter any issues that are Facebook-related, a good resource is the [official Facebook SDK for Android page](https://developers.facebook.com/android/).

Parse is compatible with v3.0 of the Facebook SDK for Android.

There are two main ways to use Facebook with your Parse users: (1) logging in as a Facebook user and creating a ParseUser, or (2) linking Facebook to an existing ParseUser.

**Login & Signup**

ParseFacebookUtils provides a way to allow your ParseUsers to log in or sign up through Facebook. This is accomplished using the logIn() method:

|  |
| --- |
| ParseFacebookUtils.logIn(this, new LogInCallback() {    @Override    public void done(ParseUser user, ParseException err) {      if (user == null) {        Log.d("MyApp", "Uh oh. The user cancelled the Facebook login.");      } else if (user.isNew()) {        Log.d("MyApp", "User signed up and logged in through Facebook!");      } else {        Log.d("MyApp", "User logged in through Facebook!");      }    }  }); |

When this code is run, the following happens:

1. The user is shown the Facebook login dialog or a prompt generated by the Facebook app.
2. The user authenticates via Facebook, and your app receives a callback.
3. Our SDK receives the Facebook data and saves it to a ParseUser. If it's a new user based on the Facebook ID, then that user is created.
4. Your LogInCallback is called with the user.

In order to display the Facebook login dialogs and activities, the current Activity must be provided (often, the current activity is this when calling logIn() from within the Activity) as we have done above.

You may optionally provide a collection of strings that specifies what read permissions your app requires from the Facebook user. You may specify these strings yourself, or use the constants we've provided for you in the ParseFacebookUtils.Permissions class. For example:

|  |
| --- |
| ParseFacebookUtils.logIn(Arrays.asList("email", Permissions.Friends.ABOUT\_ME),          this, new LogInCallback() {    @Override    public void done(ParseUser user, ParseException err) {      // Code to handle login.    }  }); |

ParseUser integration doesn't require any permissions to work out of the box (ie. null or specifying no permissions is perfectly acceptable). When logging in, you can only use read permissions. See our documentation below about [requesting additional permissions](https://parse.com/docs/android_guide#fbusers-permissions) (read or publish). [Read more about permissions on Facebook's developer guide.](https://developers.facebook.com/docs/reference/api/permissions/)

It is up to you to record any data that you need from the Facebook user after they authenticate. To accomplish this, you'll need to [do a graph query via Facebook's SDK](https://parse.com/questions/how-can-i-find-parse-users-that-are-facebook-friends-with-the-current-user).

**Linking**

If you want to associate an existing ParseUser to a Facebook account, you can link it like so:

|  |
| --- |
| if (!ParseFacebookUtils.isLinked(user)) {    ParseFacebookUtils.link(user, this, new SaveCallback() {      @Override      public void done(ParseException ex) {        if (ParseFacebookUtils.isLinked(user)) {          Log.d("MyApp", "Woohoo, user logged in with Facebook!");        }      }    });  } |

The steps that happen when linking are very similar to log in. The difference is that on successful login, the existing ParseUser is updated with the Facebook information. Future logins via Facebook will now log the user into their existing account.

If you want to unlink Facebook from a user, simply do this:

|  |
| --- |
| ParseFacebookUtils.unlinkInBackground(user, new SaveCallback() {    @Override    public void done(ParseException ex) {      if (ex == null) {        Log.d("MyApp", "The user is no longer associated with their Facebook account.");      }    }  }); |

Or if you are handling threading yourself, you can call unlink() directly like this:

|  |
| --- |
| try {    ParseFacebookUtils.unlink(user);    Log.d("MyApp", "The user is no longer associated with their Facebook account.");  } catch (ParseException e) {  } |

**Requesting Permissions**

As of v3.0 of the Facebook SDK, read and publish permissions must be requested separately. ParseFacebookUtils.logIn() and ParseFacebookUtils.link() only allow you to request read permissions. To request additional permissions, you may call ParseFacebookUtils.getSession().requestNewReadPermissions() or ParseFacebookUtils.getSession().requestNewPublishPermissions(). For more information about requesting new permissions, please see [Facebook's API documentation for these functions](https://developers.facebook.com/docs/reference/android/3.0/Session#requestNewReadPermissions%28NewPermissionsRequest%29).

After successfully retrieving new permissions, please call ParseFacebookUtilities.saveLatestSessionData(), which will save any changes to the session token back to the ParseUser and ensure that this session data follows the user wherever it logs in.

**Facebook SDK and Parse**

The Facebook Android SDK provides a number of helper classes for interacting with Facebook's API. Generally, you will use the Request class to interact with Facebook on behalf of your logged-in user. [You can read more about the Facebook SDK here](https://developers.facebook.com/docs/reference/android/3.0).

Our library manages the user's Session object for you. You can simply call ParseFacebookUtils.getSession() to access the session instance, which can then be passed to Requests.

**Twitter Users**

As with Facebook, Parse also provides an easy way to integrate Twitter authentication into your application. The Parse SDK provides a straightforward way to authorize and link a Twitter account to your ParseUsers. With just a few lines of code, you'll be able to provide a "log in with Twitter" option in your app, and be able to save their data to Parse.

**Setup**

To start using Twitter with Parse, you need to:

1. [Set up a Twitter app](https://dev.twitter.com/apps), if you haven't already.
2. Add your application's Twitter consumer key on your Parse application's settings page.
3. When asked to specify a "Callback URL" for your Twitter app, please insert a valid URL. This value will not be used by your iOS or Android application, but is necessary in order to enable authentication through Twitter.
4. Add the following where you initialize the Parse SDK in your Application.onCreate()

|  |
| --- |
| ParseTwitterUtils.initialize("YOUR CONSUMER KEY", "YOUR CONSUMER SECRET"); |

If you encounter any issues that are Twitter-related, a good resource is the [official Twitter documentation](https://dev.twitter.com/docs).

There are two main ways to use Twitter with your Parse users: (1) logging in as a Twitter user and creating a ParseUser, or (2) linking Twitter to an existing ParseUser.

**Login & Signup**

ParseTwitterUtils provides a way to allow your ParseUsers to log in or sign up through Twitter. This is accomplished using the logIn() method:

|  |
| --- |
| ParseTwitterUtils.logIn(this, new LogInCallback() {    @Override    public void done(ParseUser user, ParseException err) {      if (user == null) {        Log.d("MyApp", "Uh oh. The user cancelled the Twitter login.");      } else if (user.isNew()) {        Log.d("MyApp", "User signed up and logged in through Twitter!");      } else {        Log.d("MyApp", "User logged in through Twitter!");      }    }  }); |

When this code is run, the following happens:

1. The user is shown the Twitter login dialog.
2. The user authenticates via Twitter, and your app receives a callback.
3. Our SDK receives the Twitter data and saves it to a ParseUser. If it's a new user based on the Twitter handle, then that user is created.
4. Your LogInCallback is called with the user.

In order to display the Twitter login dialogs and activities, the current Context must be provided (often, the current context is this when calling logIn() from within the Activity) as we have done above.

**Linking**

If you want to associate an existing ParseUser with a Twitter account, you can link it like so:

|  |
| --- |
| if (!ParseTwitterUtils.isLinked(user)) {    ParseTwitterUtils.link(user, this, new SaveCallback() {      @Override      public void done(ParseException ex) {        if (ParseTwitterUtils.isLinked(user)) {          Log.d("MyApp", "Woohoo, user logged in with Twitter!");        }      }    });  } |

The steps that happen when linking are very similar to log in. The difference is that on successful login, the existing ParseUser is updated with the Twitter information. Future logins via Twitter will now log the user into their existing account.

If you want to unlink Twitter from a user, simply do this:

|  |
| --- |
| ParseTwitterUtils.unlinkInBackground(user, new SaveCallback() {    @Override    public void done(ParseException ex) {      if (ex == null) {        Log.d("MyApp", "The user is no longer associated with their Twitter account.");      }    }  }); |

**Twitter API Calls**

Our SDK provides a straightforward way to sign your API HTTP requests to the [Twitter REST API](https://dev.twitter.com/docs/api) when your app has a Twitter-linked ParseUser. To make a request through our API, you can use the Twitter singleton provided by ParseTwitterUtils:

|  |
| --- |
| HttpClient client = new DefaultHttpClient();  HttpGet verifyGet = new HttpGet(          "<https://api.twitter.com/1/account/verify_credentials.json>");  ParseTwitterUtils.getTwitter().signRequest(verifyGet);  HttpResponse response = client.execute(verifyGet); |

**Cloud Functions**

Cloud Functions can be called from Android using ParseCloud. For example, to call the Cloud Function named hello:

|  |
| --- |
| ParseCloud.callFunctionInBackground("hello", new HashMap<String, Object>(), new FunctionCallback<String>() {    void done(String result, ParseException e) {      if (e == null) {        // result is "Hello world!"      }    }  }); |

Take a look at the [Cloud Code Guide](https://parse.com/docs/cloud_code_guide?language=Android#functions-calling) to learn more about Cloud Functions.

**GeoPoints**

Parse allows you to associate real-world latitude and longitude coordinates with an object. Adding a ParseGeoPoint to a ParseObject allows queries to take into account the proximity of an object to a reference point. This allows you to easily do things like find out what user is closest to another user or which places are closest to a user.

**ParseGeoPoint**

To associate a point with an object you first need to create a ParseGeoPoint. For example, to create a point with latitude of 40.0 degrees and -30.0 degrees longitude:

|  |
| --- |
| ParseGeoPoint point = new ParseGeoPoint(40.0, -30.0); |

This point is then stored in the object as a regular field.

|  |
| --- |
| placeObject.put("location", point); |

**Geo Queries**

Now that you have a bunch of objects with spatial coordinates, it would be nice to find out which objects are closest to a point. This can be done by adding another restriction to ParseQuery using whereNear. Getting a list of ten places that are closest to a user may look something like:

|  |
| --- |
| ParseGeoPoint userLocation = (ParseGeoPoint) userObject.get("location");  ParseQuery<ParseObject> query = ParseQuery.getQuery("PlaceObject");  query.whereNear("location", userLocation);  query.setLimit(10);  query.findInBackground(new FindCallback<ParseObject>() { ... }); |

At this point nearPlaces will be an array of objects ordered by distance (nearest to farthest) from userLocation. Note that if an additional orderByAscending()/orderByDescending() constraint is applied, it will take precedence over the distance ordering.

To limit the results using distance, check out whereWithinKilometers, whereWithinMiles, and whereWithinRadians.

It's also possible to query for the set of objects that are contained within a particular area. To find the objects in a rectangular bounding box, add the whereWithinGeoBox restriction to your ParseQuery.

|  |
| --- |
| ParseGeoPoint southwestOfSF = new ParseGeoPoint(37.708813, -122.526398);  ParseGeoPoint northeastOfSF = new ParseGeoPoint(37.822802, -122.373962);  ParseQuery<ParseObject> query = ParseQuery.getQuery("PizzaPlaceObject");  query.whereWithinGeoBox("location", southwestOfSF, northeastOfSF);  query.findInBackground(new FindCallback<ParseObject>() { ... }); |

**Caveats**

At the moment there are a couple of things to watch out for:

1. Each ParseObject class may only have one key with a ParseGeoPoint object.
2. Points should not equal or exceed the extreme ends of the ranges. Latitude should not be -90.0 or 90.0. Longitude should not be -180.0 or 180.0. Attempting to set latitude or longitude out of bounds will cause an error.

**User Interface**

At the end of the day, users of your application will be interacting with Android UI components. We provide subclasses of common UI widgets to make working with Parse data easier.

**ParseQueryAdapter**

To display collections of data, we provide an implementation of Adapter. Instead of using a basic ListAdapter backed by a static array of objects, our ParseQueryAdapter provides a layer of abstraction and allows you to easily display data from one of your Parse classes in your AdapterView of choice (e.g. ListView or GridView).

To use a ParseQueryAdapter to display data in an Activity, follow the steps outlined below in your Activity's onCreate:

1. Instantiate a ParseQueryAdapter.
2. Customize it as necessary (see the below subsections for detailed instructions to display data from specific queries, change the UI of the Views to be displayed, and more).
3. Set your new Adapter on your AdapterView with setAdapter().

When the AdapterView is attached to the window, your ParseQueryAdapter will automatically fetch the first set of data. This subclass simplifies the code that would otherwise be involved with:

1. Pagination, with a row that can be tapped to load the next page.
2. Configurable downloading and displaying of remote images within rows.
3. Automatic loading and management of the Parse objects array.
4. Callbacks from major events in the data cycle.

Consider the following code, which sets up a very simple ParseQueryAdapter to display data in a ListView. You can be up and running with a functional ListView full of data with very little configuration.

|  |
| --- |
| // Inside an Activity  public void onCreate(Bundle savedInstanceState) {    super.onCreate(savedInstanceState);    // Uses a layout with a ListView (id: "listview"), which uses our Adapter.    setContentView(R.layout.main);      ParseQueryAdapter<ParseObject> adapter = new ParseQueryAdapter<ParseObject>(this, "Instrument");    adapter.setTextKey("name");    adapter.setImageKey("photo");      ListView listView = (ListView) findViewById(R.id.listview);    listView.setAdapter(adapter);  } |

This view will display a list of Instruments by name. Notice all the code that we're not writing: we can skip the logic to fetch each consecutive page of results, to manually update and maintain the backing data array, to download images in the background and set the image data on UI widgets, and to handle touch events to trigger loading the next page of results.

The ParseQueryAdapter can be configured to customize what data to use, how to display it, and what to do before and after it's been fetched. Read on to see what you can do, and how to tweak a ParseQueryAdapter to fit all of your needs.

**Customizing the Query**

By default, the simplest ParseQueryAdapter constructor takes a Context and a Parse class name. All ParseObjects in that class are then fetched and displayed in order of their createdAt timestamps.

To change this behavior, we drew from the functionality of an ArrayAdapter: but instead of taking in a vanilla array of objects to be displayed by the adapter, ParseQueryAdapter can also take a QueryFactory class which returns a ParseQuery you define. Pass that into the constructor, and the adapter will then use that query to determine which objects to fetch and display.

See below for an example setting up a ParseQueryAdapter to display only punk and metal bands with four or more members, ordered by number of records sold:

|  |
| --- |
| ParseQueryAdapter<ParseObject> adapter =    new ParseQueryAdapter<ParseObject>(this, new QueryFactory<ParseObject>() {      public ParseQuery get() {        // Here we can configure a ParseQuery to our heart's desire.        ParseQuery query = new ParseQuery("Band");        query.whereContainedIn("genre", Arrays.asList({ "Punk", "Metal" }));        query.whereGreaterThanOrEqualTo("memberCount", 4);        query.orderByDescending("albumsSoldCount");        return query;      }    }); |

**Customizing the Rows**

The default layout for the individual Views in your AdapterView is a simple LinearLayout with a ParseImageView and a TextView. If setTextKey(String) is used with the ParseQueryAdapter, its parameter will be used to select which key on your ParseObject is displayed in the TextView. Similarly, if setImageKey(String) is used, its parameter will be used to determine the image displayed in the ImageView.

One way to customize the rows is to override getItemView(ParseObject, View, ViewGroup) or getNextPageView(View, ViewGroup) and call the superclass's implementation of the appropriate method to do the heavy lifting. If you provide your own layout to the superclass's implementation, note that getItemView(ParseObject, View, ViewGroup) and getNextPageView(View, ViewGroup) expect a TextView (id: android.R.id.text1) if the textKey is set and a ParseImageView (id: android.R.id.icon) if the imageKey is set.

Here, we inflate and configure a layout of our own, with a TextView, a ParseImageView, and an extra "description" TextView (id: R.id.description):

|  |
| --- |
| @Override  public View getItemView(ParseObject object, View v, ViewGroup parent) {    if (v == null) {      v = View.inflate(getContext(), R.layout.adapter\_item, null);    }      // Take advantage of ParseQueryAdapter's getItemView logic for    // populating the main TextView/ImageView.    // The IDs in your custom layout must match what ParseQueryAdapter expects    // if it will be populating a TextView or ImageView for you.    super.getItemView(object, v, parent);      // Do additional configuration before returning the View.    TextView descriptionView = (TextView) v.findViewById(R.id.description);    descriptionView.setText(object.getString("description"));    return v;  } |

Another way to customize the rows is to have complete control over the look of the rows by overriding ParseQueryAdapter's methods and ignoring the superclass's implementation entirely. In this example, our item views are simply rows where the color is defined by the ParseObject:

|  |
| --- |
| @Override  public View getItemView(ParseObject object, View v, ViewGroup parent) {    if (v == null) {      v = View.inflate(getContext(), R.layout.adapter\_item, null);    }    v.setBackgroundColor(object.getInt("color"));    return v;  }    @Override  public View getNextPageView(View v, ViewGroup parent) {    if (v == null) {      // R.layout.adapter\_next\_page contains an ImageView with a custom graphic      // and a TextView.      v = View.inflate(getContext(), R.layout.adapter\_next\_page, null);    }    TextView textView = (TextView) v.findViewById(R.id.nextPageTextViewId);    textView.setText("Loaded " + getCount() + " rows. Get more!");    return v;  } |

**Loading Remote Images in Rows**

ParseQueryAdapter makes it simple to display remote images. By calling setImageKey(String), you can pass in a key name on your ParseObject which should contain a ParseFile containing an image to be fetched from Parse and loaded into the ParseImageView of the corresponding row.

The image will download asynchronously, and the appropriate ParseImageView will be updated in the background. As the user scrolls and rows are recycled by the adapter, images will be fetched as rows become visible and assigned ParseObjects.

You can define a placeholder image to be used when the image fetch has not yet completed. Call setPlaceholder(Drawable) on your ParseQueryAdapter to use the specified Drawable as a fallback image.

**Lifecycle Methods**

We expose two hooks in the data lifecycle of the Adapter for you to execute custom logic — right before we query Parse for your data and right after the fetched objects have been loaded from the query. These methods are particularly useful for toggling some loading UI.

An OnQueryLoadListener can be set via setOnQueryLoadListener(OnQueryLoadListener), which provides onLoading() and onLoaded(List, Exception) methods for implementation.

**Pagination**

Pagination ensures that the table only gets one page of objects at a time. You can set the number of objects are in a page by calling setObjectsPerPage(int).

The query is automatically altered to apply pagination, and a pagination row appears at the bottom of the AdapterView to allow users to load the next page.

Pagination is turned on by default. To turn it off, call setPaginationEnabled(false). With pagination turned off, the ParseQueryAdapter will use the default ParseQuery limit of 100 items.

**Auto-loading of Data**

When the AdapterView that your ParseQueryAdapter is set on is attached to a window, the ParseQueryAdapter's loadObjects() method is automatically called, triggering the fetching of the first page of results. To disable this behavior (perhaps to delay the fetching of data, or run some custom logic ahead of time), just call setAutoload(false) and call loadObjects() manually if autoload is disabled.

**Handling Errors**

Many of the methods on ParseObject, including save(), delete(), and get() will throw a ParseException on an invalid request, such as deleting or editing an object that no longer exists in the cloud, or when there is a network failure preventing communication with the Parse Cloud. You will need to catch and deal with these exceptions.

For more details, look at the [Android API](https://parse.com/docs/android).

**Security**

We strongly recommend that you build your applications to restrict access to data as much as possible. With this in mind, we recommend that you enable automatic [anonymous user creation](https://parse.com/docs/android_guide#users-anonymous) and [specify a default ACL based upon the current user](https://parse.com/docs/android_guide#users-acls) when your application is initialized. Explicitly set public writability (and potentially public readability) on an object-by-object basis in order to protect your data from unauthorized access.

Consider adding the following code to your application startup:

|  |
| --- |
| ParseUser.enableAutomaticUser();  ParseACL defaultACL = new ParseACL();  // Optionally enable public read access while disabling public write access.  // defaultACL.setPublicReadAccess(true);  ParseACL.setDefaultACL(defaultACL, true); |

Please keep secure access to your data in mind as you build your applications for the protection of both you and your users.

**Settings**

In addition to coding securely, please review the settings pages for your applications to select options that will restrict access to your applications as much as is appropriate for your needs. For example, if users should be unable to log in without a Facebook account linked to their application, disable all other login mechanisms. Specify your Facebook application IDs, Twitter consumer keys, and other such information to enable server-side validation of your users' login attempts.