



Unity Guide

[Browse](#)

Getting Started

Getting Started

If you're familiar with web frameworks like ASP.NET MVC 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.

If you haven't installed the SDK yet, please [head over to the QuickStart guide](#) to get our SDK up and running in Unity3D. Note that our SDK requires Unity version 5.2.x or higher and targets Android apps and iOS apps. You can also check out our [API Reference](#) for more detailed information about our SDK.

Parse's Unity SDK makes heavy use of a subset of the [Task-based Asynchronous Pattern](#) so that your apps remain responsive.

Adding link.xml

Create a file and name it `link.xml` under your Assets folder and put the following code to make sure Parse Unity SDK works with Unity code optimization pipeline:

```
<linker>
  <assembly fullname="UnityEngine">
    <type
      fullname="UnityEngine.iOS.NotificationServices"
      preserve="all"/>
    <type
      fullname="UnityEngine.iOS.RemoteNotification"
      preserve="all"/>
    <type fullname="UnityEngine.AndroidJavaClass"
      preserve="all"/>
  </assembly>
</linker>
```

```

<type
fullname="UnityEngine.AndroidJavaObject"
preserve="all"/>
</assembly>

<assembly fullname="Parse.Unity">
  <namespace fullname="Parse" preserve="all"/>
  <namespace fullname="Parse.Internal"
preserve="all"/>
</assembly>
</linker>

```

Want to contribute to this doc? [Edit this section.](#)

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 start with a letter, and can contain alphanumeric characters and underscores. Values can be strings, numbers, booleans, or even arrays and dictionaries - 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 an `IDictionary<string, object>`, plus the `SaveAsync` method:

```
ParseObject gameScore = new
ParseObject("GameScore");
gameScore["score"] = 1337;
gameScore["playerName"] = "Sean Plott";
Task saveTask = gameScore.SaveAsync();
```

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 Parse Cloud. Each of these fields is filled in by Parse, so they don't exist on a `ParseObject` until a save operation has completed.

Data Types

So far we've used values with type `string` and `int` assigned to fields of a `ParseObject`. Parse also supports `double`, `DateTime`, and `null`. You can also nest `IDictionary<string, T>` and `IList<T>` objects to store more structured data within a single `ParseObject`. Overall, the following types are allowed for each field in your object:

- + String => `string`
- + Number => primitive numeric values such as `double`s, `long`s, or `float`s
- + Bool => `bool`
- + Array => objects that implement `IList<T>`

- + Object => objects that implement `IDictionary<string, T>`
- + Date => `DateTime`
- + File => `ParseFile`
- + Pointer => other `ParseObject`
- + Relation => `ParseRelation`
- + Null => `null`

Some examples:

```
int number = 42;
string str = "the number is " + number;
DateTime date = DateTime.Now;
IList<object> list = new List<object> { str,
number };
IDictionary<string, object> dictionary = new
Dictionary<string, object>
{
    { "number", number },
    { "string", str }
};

var bigObject = new ParseObject("BigObject");
bigObject["myNumber"] = number;
bigObject["myString"] = str;
bigObject["myDate"] = date;
bigObject["myList"] = list;
bigObject["myDictionary"] = dictionary;
Task saveTask = bigObject.SaveAsync();
```

We do not recommend storing large pieces of binary data like images or documents on `ParseObject`. `ParseObject`s should not exceed 128 kilobytes in size. We recommend you use `ParseFile`s to store images, documents, and other types of files. You can do so by instantiating a `ParseFile` object and setting it on a field. See [Files](#) for more details.

For more information about how Parse handles data, check out our documentation on [Data](#).

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 =  
ParseObject.GetQuery("GameScore");  
query.GetAsync("xWMyZ4YEGZ").ContinueWith(t =>  
{  
    ParseObject gameScore = t.Result;  
});
```

To get the values out of the `ParseObject`, use the `Get<T>` method.

```
int score = gameScore.Get<int>("score");  
string playerName = gameScore.Get<string>  
("playerName");  
bool cheatMode = gameScore.Get<bool>  
("cheatMode");
```

Here are some examples for handling the various supported data types:

```
ParseObject bigObject = t.Result;  
int number = bigObject.Get<int>("myNumber");  
string str = bigObject.Get<string>("myString");  
DateTime date = bigObject.Get<DateTime>  
("myDate");  
byte[] data = bigObject.Get<byte[]>("myData");  
IList<object> list = bigObject.Get<List<object>>  
("myList");  
IDictionary<string, object> dictionary =  
bigObject.Get<Dictionary<string, object>>  
("myDictionary");  
Debug.Log ("Number: " + number);  
Debug.Log ("String: " + str);  
Debug.Log ("Date: " + date);  
string dataString =  
System.Text.Encoding.UTF8.GetString(data, 0,  
data.Length);  
Debug.Log ("Data: " + dataString);  
foreach (var item in list) {  
    Debug.Log ("Item: " + item.ToString());  
}  
foreach (var key in dictionary.Keys) {  
    Debug.Log ("Key: " + key + " Value: " +  
dictionary[key].ToString());  
}
```

The three special values are provided as properties:

```
string objectId = gameScore.ObjectId;  
DateTime? updatedAt = gameScore.UpdatedAt;  
DateTime? createdAt = gameScore.CreatedAt;
```

If you need to get an object's latest data from Parse, you can call the `FetchAsync` method like so:

```
Task<ParseObject> fetchTask =  
myObject.FetchAsync();
```

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.  
var gameScore = new ParseObject("GameScore")  
{  
    { "score", 1337 },  
    { "playerName", "Sean Plott" },  
    { "cheatMode", false },  
    { "skills", new List<string> { "pwnage",  
"flying" } },  
};  
gameScore.SaveAsync().ContinueWith(t =>  
{  
    // Now let's update it with some new data.  
    In this case, only cheatMode  
    // and score will get sent to the cloud.  
    playerName hasn't changed.  
    gameScore["cheatMode"] = true;  
    gameScore["score"] = 1338;  
    gameScore.SaveAsync();  
});
```

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 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");  
Task saveTask = gameScore.SaveAsync();
```

You can also increment by any amount using

```
Increment(key, amount).
```

LISTS

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

- + `AddToList` and `AddRangeToList` append the given objects to the end of an list field.
- + `AddUniqueToList` and `AddRangeUniqueToList` add only the given objects which aren't already contained in an list field to that field. The position of the insert is not guaranteed.
- + `RemoveAllFromList` removes 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.AddRangeUniqueToList("skills", new[] {  
    "flying", "kungfu" });  
Task saveTask = gameScore.SaveAsync();
```

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

Deleting Objects

To delete an object from the cloud:

```
Task deleteTask = myObject.DeleteAsync();
```

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
Task saveTask = myObject.SaveAsync();
```

Relational Data

Objects can have relationships with other objects. To model one-to-many relationships, any `ParseObject` can be used as a value in other `ParseObject`s. 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
var myPost = new ParseObject("Post")
{
    { "title", "I'm Hungry" },
    { "content", "Where should we go for lunch?" }
};

// Create the comment
var myComment = new ParseObject("Comment")
{
    { "content", "Let's do Sushirrito." }
};

// Add a relation between the Post and Comment
myComment["parent"] = myPost;

// This will save both myPost and myComment
Task saveTask = myComment.SaveAsync();
```

You can also link objects using just their `ObjectId`s like so:

```
myComment["parent"] =
ParseObject.CreateWithoutData("Post",
"1zEcyElZ80");
```

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


```
ParseObject post =  
fetchedComment.Get<ParseObject>("parent");  
Task<ParseObject> fetchTask =  
post.FetchIfNeededAsync();
```

For a many-to-many relationship, use the `ParseRelation` object. This works similar to a `List<ParseObject>`, except that you don't need to download all the objects in a relation at once. This allows `ParseRelation` to scale to many more objects than the `List<ParseObject>` approach. For example, a `ParseUser` may have many `Post`s that they might like. In this case, you can store the set of `Post`s that a `ParseUser` likes using `GetRelation`. In order to add a post to the list, the code would look something like:

```
var user = ParseUser.CurrentUser;  
var relation = user.GetRelation<ParseObject>  
("likes");  
relation.Add(post);  
Task saveTask = user.SaveAsync();
```

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 `Post`s by using calling `FindAsync` on the `ParseQuery` returned by `Query`. The code would look like:

```
relation.Query.FindAsync().ContinueWith(t =>  
{  
    IEnumerable<ParseObject> relatedObjects =  
t.Result;  
});
```

If you want only a subset of the `Post`s you can add extra constraints to the `ParseQuery` returned by `Query` like this:

```
var query = relation.Query  
    .WhereGreaterThan("createdAt", DateTime.Now -  
    TimeSpan.FromDays(10));
```

```
// alternatively, add any other query
constraints
query.FindAsync().ContinueWith(t =>
{
    IEnumerable<ParseObject> relatedObjects =
t.Result;
});
```

For more details on `ParseQuery` please look at the [query](#) portion of this guide. A `ParseRelation` behaves similar to a `List<ParseObject>`, so any queries you can do on lists of objects you can do on `ParseRelation`s.

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<T>()`. In mature codebases, subclasses have many advantages, including terseness, extensibility, type-safety, and support for code completion. Subclassing is completely optional, but can transform this code:

```
// Using dictionary-initialization syntax:
var shield = new ParseObject("Armor")
{
    { "displayName", "Wooden Shield" },
    { "fireproof", false },
    { "rupees", 50 }
};

// And later:
Debug.Log(shield.Get<string>("displayName"));
shield["fireproof"] = true;
shield["rupees"] = 500;
```

Into this:

```
// Using object-initialization syntax:
var shield = new Armor
{
    DisplayName = "Wooden Shield",
    IsFireproof = false,
    Rupees = 50
};

// And later:
Debug.Log(shield.DisplayName);
shield.IsFireproof = true;
shield.Rupees = 500;
```

To create a `ParseObject` subclass:

- 1 Declare a subclass which extends `ParseObject`.
- 2 Add a `ParseClassName` attribute. 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>()` in a `MonoBehaviour`'s `Awake` method and attach this to your Parse initialization `GameObject`.

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

```
// Armor.cs
using Parse;

[ParseClassName("Armor")]
public class Armor : ParseObject
{
}

// ExtraParseInitialization.cs (attach to your
// Parse
// initialization GameObject)
using UnityEngine;
using Parse;

public class ExtraParseInitialization :
MonoBehaviour
{
    void Awake()
    {
        ParseObject.RegisterSubclass<Armor>();
    }
}
```

PROPERTIES AND METHODS

Adding methods and properties 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 properties 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 `GetProperty<T>()` and `SetProperty<T>()`. Finally, add a `ParseFieldName` attribute to the property to fully integrate the property with Parse, enabling functionality like automatically raising `INotifyPropertyChanged` notifications for your objects. The following example creates a `displayName` field in the `Armor` class:

```
// Armor.cs
[ParseClassName("Armor")]
public class Armor : ParseObject
{
    [ParseFieldName("displayName")]
    public string DisplayName
    {
        get { return GetProperty<string>
("DisplayName"); }
        set { SetProperty<string>(value,
"DisplayName"); }
    }
}
```

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

`ParseRelation`-typed properties can also be easily defined using `GetRelationProperty<T>`. For example:

```
// Armor.cs
[ParseClassName("Armor")]
public class Armor : ParseObject
{
    [ParseFieldName("attributes")]
    public ParseRelation<ArmorAttribute> Attributes
    {
        get { return
GetRelationProperty<ArmorAttribute>
("Attributes"); }
    }
}
```

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
    this.Increment("durability", -amount);
    if (this.Durability < 0) {
        this.IsBroken = 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<T>():
```

```
var armorReference =
ParseObject.CreateWithoutData<Armor>
(armor.ObjectId);
```

QUERIES

You can get a query for objects of a particular subclass using the generic `ParseQuery<T>` class. The following example queries for armors that the user can afford:

```
var query = new ParseQuery<Armor>()
    .WhereLessThanOrEqualTo("rupees",
((Player)ParseUser.CurrentUser).Rupees);
query.FindAsync().ContinueWith(t =>
{
    IEnumerable<Armor> result = t.Result;
});
```

Want to contribute to this doc? [Edit this section.](#)

Queries

We've already seen how a `ParseQuery` with `GetAsync` can retrieve a single `ParseObject` from Parse. There are many other ways to retrieve data with `ParseQuery`

- you can retrieve many objects at once, put conditions on the objects you wish to retrieve, and more.

Basic Queries

In many cases, `GetAsync` 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`, constraints to it, and then retrieve an `IEnumerable` of matching `ParseObjects`s using `FindAsync`.

For example, to retrieve scores with a particular `playerName`, use a "where" clause to constrain the value for a key.

```
var query = ParseObject.GetQuery("GameScore")
    .WhereEqualTo("playerName", "Dan Stemkoski");
query.FindAsync().ContinueWith(t =>
{
    IEnumerable<ParseObject> results = t.Result;
});
```

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 a call to `WhereNotEqualTo`:

```
var query = ParseObject.GetQuery("GameScore")
    .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.

```
var query = ParseObject.GetQuery("GameScore")
    .WhereNotEqualTo("playerName", "Michael
Yabuti")
    .WhereGreaterThan("playerAge", 18);
```

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

```
query = query.Limit(10); // limit to at most 10 results
```

If you want exactly one result, a more convenient alternative may be to use `FirstAsync` or `FirstOrDefaultAsync` instead of using `FindAsync`.

```
var query = ParseObject.GetQuery("GameScore")
    .WhereEqualTo("playerEmail",
        "dstemkoski@example.com");
query.FirstAsync().ContinueWith(t =>
{
    ParseObject obj = t.Result;
});
```

You can skip the first results by calling `Skip`. This can be useful for pagination:

```
query = query.Skip(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 score
// and descending order by playerName
var query = ParseObject.GetQuery("GameScore")
    .OrderBy("score")
    .ThenByDescending("playerName");
```

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

```
// Restricts to wins < 50
query = query.WhereLessThan("wins", 50);

// Restricts to wins <= 50
query = query.WhereLessThanOrEqualTo("wins",
50);

// Restricts to wins > 50
query = query.WhereGreaterThan("wins", 50);

// Restricts to wins >= 50
query = query.WhereGreaterThanOrEqualTo("wins",
50);
```

If you want to retrieve objects matching several different values, you can use `WhereContainedIn`, providing an list 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:

```
// Finds scores from any of Jonathan, Dario, or
Shawn
var names = new[] { "Jonathan Walsh", "Dario
Wunsch", "Shawn Simon" };
var query = ParseObject.GetQuery("GameScore")
    .WhereContainedIn("playerName", names);
```

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

```
// Finds scores from any of Jonathan, Dario, or
Shawn
var names = new[] { "Jonathan Walsh", "Dario
Wunsch", "Shawn Simon" };
var query = ParseObject.GetQuery("GameScore")
    .WhereNotContainedIn("playerName", 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
var query = ParseObject.GetQuery("GameScore")
    .WhereExists("score");

// Finds objects that don't have the score set
var query = ParseObject.GetQuery("GameScore")
    .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:


```

var teamQuery = ParseQuery.GetQuery("Team")
    .WhereGreaterThan("winPct", 0.5);
var userQuery = ParseUser.Query
    .WhereMatchesKeyInQuery("hometown", "city",
teamQuery);
userQuery.FindAsync(t =>
{
    IEnumerable<ParseUser> results = t.Result;
});
// results will contain users with a hometown
team with a winning record

```

Queries on List Values

For keys with an array type, you can find objects where the key's array value contains 2 by:

```

// Find objects where the list in listKey
contains 2.
var query = ParseObject.GetQuery("MyClass")
    .WhereEqualTo("listKey", 2);

```

Queries on String Values

If you're trying to implement a generic search feature, we recommend taking a look at this blog post:

[Implementing Scalable Search on a NoSQL Backend.](#)

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".
var query = ParseObject.GetQuery("BarbecueSauce")
    .WhereStartsWith("name", "Big Daddy's");

```

The above example will match any `BarbecueSauce` objects where the value in the “name” String key starts with “Big Daddy’s”. For example, both “Big Daddy’s” and “Big Daddy’s BBQ” will match, but “big daddy’s” or “BBQ Sauce: Big Daddy’s” will not.

Queries that have regular expression constraints are very expensive. Refer to the [Performance Guide](#) for

more details.

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.
var query = ParseObject.GetQuery("Comment")
    .WhereEqualTo("post", myPost);

query.FindAsync().ContinueWith(t =>
{
    IEnumerable<ParseObject> comments = t.Result;
    // comments now contains the comments for
    myPost
});
```

You can also do relational queries by `ObjectId`:

```
var query = ParseObject.GetQuery("Comment")
    .WhereEqualTo("post",
ParseObject.CreateWithoutData("Post",
"1zEcyElZ80"));
```

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 with images, you can do:

```
var imagePosts = ParseObject.GetQuery("Post")
    .WhereExists("image");
var query = ParseObject.GetQuery("Comment")
    .WhereMatchesQuery("post", imagePosts);

query.FindAsync().ContinueWith(t =>
{
    IEnumerable<ParseObject> comments = t.Result;
    // 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:

```
var imagePosts = ParseObject.GetQuery("Post")  
    .WhereExists("image");  
var query = ParseObject.GetQuery("Comment")  
    .WhereDoesNotMatchQuery("post", imagePosts);  
  
query.FindAsync().ContinueWith(t =>  
{  
    IEnumerable<ParseObject> comments = t.Result;  
    // 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:

```
// Retrieve the most recent comments  
var query = ParseObject.GetQuery("Comment")  
    .OrderByDescending("createdAt")  
    .Limit(10) // Only retrieve the last 10  
comments  
    .Include("post"); // Include the post data  
with each comment  
  
// Only retrieve the last 10 comments  
query = query.Limit(10);  
  
// Include the post data with each comment  
query = query.Include("post");  
  
query.FindAsync().ContinueWith(t =>  
{  
    IEnumerable<ParseObject> comments = t.Result;  
  
    // Comments now contains the last ten  
    comments, and the "post" field  
    // contains an object that has already been  
    fetched. For example:  
    foreach (var comment in comments)  
    {  
        // This does not require a network  
        access.  
        var post = comment.Get<ParseObject>  
("post");  
        Debug.Log("Post title: " +
```

```
post["title"]);  
    }  
});
```

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 = 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 `FirstAsync` and `GetAsync`

Counting Objects

Caveat: Count queries are rate limited to a maximum of 160 requests per minute. They can also return inaccurate results for classes with more than 1,000 objects. Thus, it is preferable to architect your application to avoid this sort of count operation (by using counters, for example.)

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

```
// First set up a callback.  
var query = ParseObject.GetQuery("GameScore")  
    .WhereEqualTo("playerName", "Sean Plott");  
query.CountAsync().ContinueWith(t =>  
{  
    int count = t.Result;  
});
```

Compound Queries

If you want to find objects that match one of several queries, you can use the `Or` method. For instance, if you want to find players that either have a lot of wins or a few wins, you can do:

```
var lotsOfWins = ParseObject.GetQuery("Player")  
    .WhereGreaterThan("wins", 150);
```

```
var fewWins = ParseObject.GetQuery("Player")
    .WhereLessThan("wins", 5);

ParseQuery<ParseObject> query =
lotsOfWins.Or(fewWins);
// results contains players with lots of wins or
only a few wins.
```

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

Note that we do not, however, support `GeoPoint` or non-filtering constraints (e.g. `Near`, `WhereWithinGeoBox`, `Limit`, `Skip`, `OrderBy...`, `Include`) in the subqueries of the compound query.

Want to contribute to this doc? [Edit this section.](#)

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 `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` through these properties, you do not need to set it using the indexer on `ParseObject` — this is set for you automatically.

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:

```
var user = new ParseUser()
{
    Username = "my name",
    Password = "my pass",
    Email = "email@example.com"
};

// other fields can be set just like with
ParseObject
user["phone"] = "415-392-0202";

Task signUpTask = user.SignUpAsync();
```

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

Note that we used the `SignUpAsync` method, not the `SaveAsync` method. New `ParseUser`s should always be created using the `SignUpAsync` method. Subsequent updates to a user can be done by calling `SaveAsync`.

If a signup isn't successful, you should catch the exception thrown by the `SignUpAsync`. 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 address as the username. Simply ask your users to enter their email, but fill it in

both the `Username` and `Email` properties — `ParseObject` 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 to let them log in to their account in the future. To do this, you can use the class method `LogInAsync`.

```
ParseUser.LogInAsync("myname",
"mypass").ContinueWith(t =>
{
    if (t.IsFaulted || t.IsCanceled)
    {
        // The login failed. Check the error to
see why.
    }
    else
    {
        // Login was successful.
    }
})
```

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. `ParseUser`s can never have a `true` value when the user account is first created.
- 2 `false` - at the time the `ParseUser` object was last refreshed, the user had not confirmed his or her email address. If `emailVerified` is `false`, consider calling `FetchAsync` 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 `ParseUser.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:

```
if (ParseUser.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();
var currentUser = ParseUser.CurrentUser; // this
will now be null
```

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 the `SaveAsync` or `DeleteAsync` methods unless the `ParseUser` was obtained using an authenticated method, like `LogInAsync` or `SignUpAsync`. This ensures that only the user can alter their own data.

The following illustrates this security policy:

```
ParseUser user = null;
ParseUser.LogInAsync("my_username",
"my_password").ContinueWith(t =>
{
    user = t.Result;
    user.Username = "my_new_username"; // attempt
to change username
    return user.SaveAsync();
}).Unwrap().ContinueWith(t =>
```



```

    }

    if (!t.IsFaulted)
    {
        // This succeeds, since this user was
        authenticated
        // on the device

        ParseUser.LogOut();
    }
}).ContinueWith(t =>
{
    // Get the user from a non-authenticated
    method
    return
    ParseUser.Query.GetAsync(user.ObjectId);
}).Unwrap().ContinueWith(t =>
{
    user = t.Result;
    user.Username = "another_username";

    return user.SaveAsync();
}).Unwrap().ContinueWith(t =>
{
    if (t.IsFaulted)
    {
        // This will always fail, since the
        ParseUser is not authenticated
    }
});
}

```

The `ParseUser` obtained from `Current` will always be authenticated.

If you need to check if a `ParseUser` is authenticated, you can check the `IsAuthenticated` property. 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**, 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, the `ParseACL` constructor 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:

```
var privateNote = new ParseObject("Note");
privateNote["content"] = "This note is private!";
privateNote.ACL = new
ParseACL(ParseUser.CurrentUser);
Task saveTask = privateNote.SaveAsync();
```

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:

```
var groupMessage = new ParseObject("Message");
var groupACL = new ParseACL();

// userList is an IEnumerable<ParseUser> with the
// users we are sending
// this message to.
foreach (var user in userList)
{
    groupACL.SetReadAccess(user, true);
    groupACL.SetWriteAccess(user, true);
}

groupMessage.ACL = groupACL;
Task saveTask = groupMessage.SaveAsync();
```

You can also grant permissions to all users at once using the `PublicReadAccess` and `PublicWriteAccess` properties. 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:

```
var publicPost = new ParseObject("Post");
var postACL = new ParseACL(ParseUser.CurrentUser)
{
    PublicReadAccess = true,
```

```
PublicWriteAccess = false
};
publicPost.ACL = postACL;
Task saveTask = publicPost.SaveAsync();
```

Operations that are forbidden, such as deleting an object that you do not have write access to, result in a `ParseException` with an `OtherCause` 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

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:

```
Task requestPasswordTask =
ParseUser.RequestPasswordResetAsync("email@example
```

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:

```
ParseUser.Query
    .WhereEqualTo("gender", "female")
    .FindAsync().ContinueWith(t =>
    {
        IEnumerable<ParseUser> women = t.Result;
    });
```

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

Associations

Associations involving a `ParseUser` work right out 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:

```
// Make a new post
var post = new ParseObject("Post")
{
    { "title", "My New Post" },
    { "body", "This is some great content." },
    { "user", ParseUser.CurrentUser }
};
post.SaveAsync().ContinueWith(t =>
{
    // Find all posts by the current user
    return ParseObject.GetQuery("Post")
        .WhereEqualTo("user",
ParseUser.CurrentUser)
        .FindAsync();
}).Unwrap().ContinueWith(t =>
{
    IEnumerable<ParseObject> userPosts =
t.Result;
});
```

Facebook Users

Parse provides an easy way to integrate Facebook with your application. The `ParseFacebookUtils` class integrates with `ParseUser` 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**, if you haven't already. In the "Advanced" tab of your app's settings page, Make sure that your app's "App Type" (in the "Authentication" section) is set to "Native/Desktop" and "Is your App Secret embedded?" is set to No.
- 2 Add your application's Facebook Application ID on your Parse application's settings page.

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 `ParseUser`s to log in or sign up through Facebook. This is accomplished using the `LogInAsync()` method. Use the Facebook Unity SDK to log into Facebook, then call `LogInAsync()` with the access token information:

```
Task<ParseUser> logInTask =  
ParseFacebookUtils.LogInAsync(userId,  
accessToken, tokenExpiration);
```

When this code is run, 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.

When this code is run, the following happens:

- 1 The user is shown the Facebook login dialog.
- 2 The user authenticates via Facebook, and your app receives a callback.
- 3 Our SDK receives the user's Facebook access data and saves it to a `ParseUser`. If no `ParseUser` exists with the same Facebook ID, then a new `ParseUser` is created.

- 4 The current user reference will be updated to this user.

LINKING

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

```
if (!ParseFacebookUtils.IsLinked(user))
{
    Task linkTask =
ParseFacebookUtils.LinkAsync(user, userId,
accessToken, tokenExpiration);
}
```

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 a Facebook account from a user, simply do this:

```
Task unlinkTask =
ParseFacebookUtils.UnlinkAsync(user);
```

Want to contribute to this doc? [Edit this section.](#)

Sessions

Sessions represent an instance of a user logged into a device. Sessions are automatically created when users log in or sign up. They are automatically deleted when users log out. There is one distinct `Session` object for each user-installation pair; if a user issues a login request from a device they're already logged into, that user's previous `Session` object for that Installation is automatically deleted. `Session` objects are stored on Parse in the `Session` class, and you can view them on the Parse.com Data Browser. We provide a set of APIs to manage `Session` objects in your app.

`Session` is a subclass of a Parse `Object`, so you can query, update, and delete sessions in the same way that

you manipulate normal objects on Parse. Because the Parse Cloud automatically creates sessions when you log in or sign up users, you should not manually create `Session` objects unless you are building a “Parse for IoT” app (e.g. Arduino or Embedded C). Deleting a `Session` will log the user out of the device that is currently using this session’s token.

Unlike other Parse objects, the `Session` class does not have Cloud Code triggers. So you cannot register a `beforeSave` or `afterSave` handler for the `Session` class.

Properties

The `Session` object has these special fields:

- + `sessionToken` (readonly): String token for authentication on Parse API requests. In the response of `Session` queries, only your current `Session` object will contain a session token.
- + `user`: (readonly) Pointer to the `User` object that this session is for.
- + `createdWith` (readonly): Information about how this session was created (e.g. `{ "action": "login", "authProvider": "password" }`).
 - + `action` could have values: `login`, `signup`, `create`, or `upgrade`. The `create` action is when the developer manually creates the session by saving a `Session` object. The `upgrade` action is when the user is upgraded to revocable session from a legacy session token.
 - + `authProvider` could have values: `password`, `anonymous`, `facebook`, or `twitter`.
- + `restricted` (readonly): Boolean for whether this session is restricted.
 - + Restricted sessions do not have write permissions on `User`, `Session`, and `Role` classes on Parse. Restricted sessions also cannot read unrestricted sessions.
 - + All sessions that the Parse Cloud automatically creates during user login/signup will be unrestricted. All sessions that the developer manually

creates by saving a new `Session` object from the client (only needed for “Parse for IoT” apps) will be restricted.

- + `expiresAt` (readonly): Approximate UTC date when this `Session` object will be automatically deleted. You can configure session expiration settings (either 1-year inactivity expiration or no expiration) in your app’s Parse.com dashboard settings page.
- + `installationId` (can be set only once): String referring to the `Installation` where the session is logged in from. For Parse SDKs, this field will be automatically set when users log in or sign up. All special fields except `installationId` can only be set automatically by the Parse Cloud. You can add custom fields onto `Session` objects, but please keep in mind that any logged-in device (with session token) can read other sessions that belong to the same user (unless you disable Class-Level Permissions, see below).

Handling Invalid Session Token Error

Apps created on Parse.com before March 25, 2015 use legacy session tokens until you [migrate them to use the new revocable sessions](#). On API requests with legacy tokens, if the token is invalid (e.g. User object was deleted), then the request is executed as a non-logged in user and no error was returned. On API requests with revocable session tokens, an invalid session token will always fail with the “invalid session token” error. This new behavior lets you know when you need to ask the user to log in again.

With revocable sessions, your current session token could become invalid if its corresponding `Session` object is deleted from the Parse Cloud. This could happen if you implement a Session Manager UI that lets users log out of other devices, or if you manually delete the session via Cloud Code, REST API, or Data Browser. Sessions could also be deleted due to automatic expiration (if configured in app settings). When a device’s session token no longer corresponds to a `Session` object on the Parse Cloud, all API requests from that device will fail with “Error 209: invalid session token”.

To handle this error, we recommend writing a global utility function that is called by all of your Parse request error callbacks. You can then handle the “invalid session

token” error in this global function. You should prompt the user to login again so that they can obtain a new session token. This code could look like this:

```
public class ParseErrorHandler {
    public static void
    HandleParseError(ParseException e) {
        switch (e.Code) {
            case
            ParseException.ErrorCode.InvalidSessionToken:
                HandleInvalidSessionToken()
                break;

            ... // Other Parse API errors that you want
            to explicitly handle
        }
    }

    private static void HandleInvalidSessionToken()
    {
        //-----
        // Option 1: Show a message asking the user
        to log out and log back in.
        //-----
        // If the user needs to finish what they were
        doing, they have the opportunity to do so.

        //-----
        // Option #2: Show login screen so user can
        re-authenticate.
        //-----
        // You may want this if the logout button is
        inaccessible in the UI.
    }
}

// In all API requests, call the global error
// handler, e.g.
query.FindAsync().ContinueWith(t => {
    if (t.IsFaulted) {
        // Query Failed - handle an error.

        ParseErrorHandler.HandleParseError(t.Exception.InnerException
        as ParseException);
    } else {
        // Query Succeeded - continue your app logic
        here.
    }
});
```

Security

`Session` objects can only be accessed by the user specified in the user field. All `Session` objects have an ACL that is read and write by that user only. You cannot change this ACL. This means querying for sessions will only return objects that match the current logged-in user.

When you log in a user via a `User` login method, Parse will automatically create a new unrestricted `Session` object in the Parse Cloud. Same for signups and Facebook/Twitter logins.

Session objects manually created from client SDKs (by creating an instance of `Session`, and saving it) are always restricted. You cannot manually create an unrestricted sessions using the object creation API.

Restricted sessions are prohibited from creating, modifying, or deleting any data in the `User`, `Session`, and `Role` classes. Restricted session also cannot read unrestricted sessions. Restricted Sessions are useful for “Parse for IoT” devices (e.g Arduino or Embedded C) that may run in a less-trusted physical environment than mobile apps. However, please keep in mind that restricted sessions can still read data on `User`, `Session`, and `Role` classes, and can read/write data in any other class just like a normal session. So it is still important for IoT devices to be in a safe physical environment and ideally use encrypted storage to store the session token.

If you want to prevent restricted Sessions from modifying classes other than `User`, `Session`, or `Role`, you can write a Cloud Code `beforeSave` handler for that class:

```
Parse.Cloud.beforeSave("MyClass",
function(request, response) {
  Parse.Session.current().then(function(session)
  {
    if (session.get('restricted')) {
      response.error('write operation not
allowed');
    }
    response.success();
  });
});
```

You can configure Class-Level Permissions (CLPs) for the Session class just like other classes on Parse. CLPs restrict reading/writing of sessions via the `Session` API, but do not restrict Parse Cloud's automatic session creation/deletion when users log in, sign up, and log out. We recommend that you disable all CLPs not

needed by your app. Here are some common use cases for Session CLPs:

- + **Find, Delete** — Useful for building a UI screen that allows users to see their active session on all devices, and log out of sessions on other devices. If your app does not have this feature, you should disable these permissions.
- + **Create** — Useful for “Parse for IoT” apps (e.g. Arduino or Embedded C) that provision restricted user sessions for other devices from the phone app. You should disable this permission when building apps for mobile and web. For “Parse for IoT” apps, you should check whether your IoT device actually needs to access user-specific data. If not, then your IoT device does not need a user session, and you should disable this permission.
- + **Get, Update, Add Field** — Unless you need these operations, you should disable these permissions.

Want to contribute to this doc? [Edit this section.](#)

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**. 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, must be unique, 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** to the set of users that will inherit permissions granted to the containing role.
- + **roles**: A **relation** 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.
var roleACL = new ParseACL();
roleACL.PublicReadAccess = true;
var role = new ParseRole("Administrator",
roleACL);
Task saveTask = role.SaveAsync();
```

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

```
var role = new ParseRole(roleName, roleACL);
foreach (ParseUser user in usersToAddToRole)
{
    role.Users.Add(user);
}
foreach (ParseRole childRole in rolesToAddToRole)
{
    role.Roles.Add(childRole);
}
Task saveTask = role.SaveAsync();
```

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.Query
    .WhereEqualTo("name", "Moderators")
    .FirstAsync()
    .ContinueWith(t =>
    {
        var moderators = t.Result;
        var wallPost = new
ParseObject("WallPost");
        var postACL = new ParseACL();
        postACL.SetRoleWriteAccess(moderators,
true);
        wallPost.ACL = postACL;
        return wallPost.SaveAsync();
    });
```

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

```
var wallPost = new ParseObject("WallPost");
var postACL = new ParseACL();
postACL.SetRoleWriteAccess("Moderators", true);
wallPost.ACL = postACL;
Task saveTask = wallPost.SaveAsync();
```

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.Roles.Add(administrators);
Task saveTask = moderators.SaveAsync();
```

Want to contribute to this doc? [Edit this section.](#)

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[]` or `Stream` form and then create a `ParseFile` with it. In this example, we'll just use a string:

```
byte[] data =  
System.Text.Encoding.UTF8.GetBytes("Working at  
Parse is great!");  
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`, you can call `SaveAsync` to save the file to Parse.

```
Task saveTask = file.SaveAsync();
```

Finally, after the save completes, you can assign a `ParseFile` into a `ParseObject` just like any other piece of data:

```
var jobApplication = new  
ParseObject("JobApplication");  
jobApplication["applicantName"] = "Joe Smith";  
jobApplication["applicantResumeFile"] = file;  
Task saveTask = jobApplication.SaveAsync();
```

Retrieving it back involves downloading the resource at the `ParseFile`'s `Url`. Here we retrieve the resume file off another `JobApplication` object:

```
var applicantResumeFile =  
anotherApplication.Get<ParseFile>  
("applicantResumeFile");  
var resumeTextRequest = new  
WWW(applicantResumeFile.Url.AbsoluteUri);  
yield return resumeTextRequest;  
string resumeText = resumeTextRequest.text;  
</code></pre>
```

Progress

It's easy to get the progress of `ParseFile` uploads by passing a `Progress` object to `SaveAsync`. For example:

```
```cs  
byte[] data =
System.Text.Encoding.UTF8.GetBytes("Working at
Parse is great!");
ParseFile file = new ParseFile("resume.txt",
data);

Task saveTask = file.SaveAsync(new
Progress<ParseUploadProgressEventArgs>(e => {
 // Check e.Progress to get the progress of
 the file upload
}));
```
```

You can delete files that are referenced by objects using the [REST API](#). You will need to provide the master key in order to be allowed to delete a file.

If your files are not referenced by any object in your app, it is not possible to delete them through the REST API. You may request a cleanup of unused files in your app's Settings page. Keep in mind that doing so may break functionality which depended on accessing unreferenced files through their URL property. Files that are currently associated with an object will not be affected.

Want to contribute to this doc? [Edit this section](#).

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:

```
var point = new ParseGeoPoint(40.0, -30.0);
```

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

```
placeObject["location"] = point;
```

Note: Currently only one key in a class may be a `ParseGeoPoint`.

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 a `ParseQuery` using `WhereNear`. Getting a list of ten places that are closest to a user may look something like:

```
// User's location
var userGeoPoint =
ParseUser.CurrentUser.Get<ParseGeoPoint>
("location");
// Create a query for places
var query = ParseObject.GetQuery("PlaceObject");
//Interested in locations near user.
query = query.WhereNear("location",
userGeoPoint);
// Limit what could be a lot of points.
query = query.Limit(10);
// Final list of nearby places
query.FindAsync().ContinueWith(t =>
{
    IEnumerable<ParseObject> nearbyPlaces =
```

```
t.Result;  
});
```

At this point `placesObjects` will be an `IEnumerable<ParseObject>` of `PlaceObject`s ordered by distance (nearest to farthest) from `userGeoPoint`.

To limit the results using distance check out `WhereWithinDistance`.

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`.

```
var swOfSF = new ParseGeoPoint(37.708813,  
-122.526398);  
var neOfSF = new ParseGeoPoint(37.822802,  
-122.373962);  
var query =  
ParseObject.GetQuery("PizzaPlaceObject")  
    .WhereWithinGeoBox("location", swOfSF,  
neOfSF);  
query.FindAsync().ContinueWith(t =>  
{  
    IEnumerable<ParseObject> pizzaPlacesInSF =  
t.Result;  
});
```

Geo Distances

Parse makes it easy to find the distance between two `GeoPoints` and query based upon that distance. For example, to get a distance in kilometers between two points, you can use the `DistanceTo` method:

```
ParseGeoPoint p1 = /* Some location */;  
ParseGeoPoint p2 = /* Some other location */;  
double distanceInKm =  
p1.DistanceTo(p2).Kilometers;
```

You can also query for `ParseObject`s within a radius using a `ParseGeoDistance`. For example, to find all places within 5 miles of a user, you would use the `WhereWithinDistance` method:

```

ParseGeoPoint userGeoPoint =
ParseUser.CurrentUser.Get<ParseGeoPoint>
("location");
ParseQuery<ParseObject> query =
ParseObject.GetQuery("PlaceObject")
    .WhereWithinDistance("location",
userGeoPoint, ParseGeoDistance.FromMiles(5));
query.FindAsync().ContinueWith(t =>
{
    IEnumerable<ParseObject> nearbyLocations =
t.Result;
    // nearbyLocations contains PlaceObjects
within 5 miles of the user's location
});

```

At this point, `nearbyLocations` will be an array of objects ordered by distance (nearest to farthest) from `userGeoPoint`. Note that if an additional `OrderBy()` constraint is applied, it will take precedence over the distance ordering.

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 Using the `WhereNear` constraint will also limit results to within 100 miles.
- 3 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.

Want to contribute to this doc? [Edit this section.](#)

Push Notifications

Push Notifications are a great way to keep your users engaged and informed about your app. You can reach your entire user base quickly and effectively. This guide will help you through the setup process and the general usage of Parse to send push notifications.

If you haven't installed the SDK yet, please [head over to the Push QuickStart](#) to get our SDK up and running.

The Unity SDK can send push notifications from all runtimes, but only iOS and Android apps can receive pushes from APNS and GCM push servers.

Setting Up Push

Currently, only two platforms are supported to receive push from Parse, Unity iOS and Unity Android. * If you want to start using push on Unity iOS, start by completing the [iOS Push QuickStart](#) to learn how to configure your push certificate. * If you want to start using push on Unity Android, start by completing [Android Push QuickStart](#) to learn how to configure your app. Come back to this guide afterwards to learn more about the push features offered by Parse.

Installations

Every Parse application installed on a device registered for push notifications has an associated `Installation` object. The `Installation` object is where you store all the data needed to target push notifications. For example, in a baseball app, you could store the teams a user is interested in to send updates about their performance. Saving the `Installation` object is also required for tracking push-related app open events.

On Unity, `Installation` objects are available through the `ParseInstallation` class, a subclass of `ParseObject`. It uses the [same API](#) for storing and retrieving data. To access the current `Installation` object from your .NET app, use the `ParseInstallation.CurrentInstallation` property.

While it is possible to modify a `ParseInstallation` just like you would a `ParseObject`, there are several special fields that help manage and target devices.

- + `channels`: An `IEnumerable<string>` of the channels to which a device is currently subscribed. In .NET, this field is accessible through the `Channels` property.
- + `timeZone`: The current time zone where the target device is located. This field is readonly and

can be accessed via the `TimeZone` property. This value is synchronized every time an `Installation` object is saved from the device.

- + `localeIdentifier`: The locale identifier of the device in the format [language code]-[COUNTRY CODE]. The language codes are two-letter lowercase ISO language codes (such as “en”) as defined by [ISO 639-1](#). The country codes are two-letter uppercase ISO country codes (such as “US”) as defined by [ISO 3166-1](#). This value is synchronized every time a `ParseInstallation` object is saved from the device (*readonly*).
- + `deviceType`: The type of device, “ios”, “android”, “winrt”, “winphone”, or “dotnet”. This field is readonly and can be accessed via the `DeviceType` property.
- + `pushType`: This field is reserved for directing Parse to the push delivery network to be used. If the device is registered to receive pushes via GCM, this field will be marked “gcm”. If this device is not using GCM, and is using Parse’s push notification service, it will be blank (*readonly*).
- + `installationId`: Unique Id for the device used by Parse. This field is readonly and can be accessed via the `InstallationId` property.
- + `installationId`: Unique Id for the device used by Parse. This field is readonly and can be accessed via the `InstallationId` property.
- + `channelUri`: The Microsoft-generated push URIs for Windows devices. This field is readonly and can be accessed via the `DeviceUri` property.
- + `appName`: The display name of the client application to which this installation belongs. This field is readonly and can be accessed via the `AppName` property.
- + `appVersion`: The version string of the client application to which this installation belongs. This field is readonly and can be accessed via the `AppVersion` property.
- + `parseVersion`: The version of the Parse SDK which this installation uses. This field is readonly and can be accessed via the `ParseVersion` property.
- + `appIdIdentifier`: A unique identifier for this installation’s client application. This field is readonly and can be accessed via the `AppIdentifier` property. On Windows 8, this is the `Windows.ApplicationModel.Package` id; on Windows Phone 8 this is the `ProductId`; in other

.NET applications, this is the `ApplicationIdentity` of the current `AppDomain`

Sending Pushes

There are two ways to send push notifications using Parse: **channels** and **advanced targeting**. Channels offer a simple and easy to use model for sending pushes, while advanced targeting offers a more powerful and flexible model. Both are fully compatible with each other and will be covered in this section.

Sending notifications is often done from the Parse.com push console, the **REST API** or from **Cloud Code**.

However, push notifications can also be triggered by the existing client SDKs. If you decide to send notifications from the client SDKs, you will need to set **Client Push Enabled** in the Push Notifications settings of your Parse app.

However, be sure you understand that enabling Client Push can lead to a security vulnerability in your app, as outlined [on our blog](#). We recommend that you enable Client Push for testing purposes only, and move your push notification logic into Cloud Code when your app is ready to go into production.

Push Notification Settings

Secure push notifications for your app.

Enable client push?
Allow pushes to be sent using the public client keys. Useful during development, but we suggest disabling it on production apps.

No ☒ Yes

Enable REST push?
Allow push notifications to be sent using the REST key. When enabled, be sure to keep your REST key secret.

No ☒ Yes

You can view your past push notifications on the Parse.com push console for up to 30 days after creating your push. For pushes scheduled in the future, you can delete the push on the push console as long as no sends have happened yet. After you send the push, the push console shows push analytics graphs.

USING CHANNELS

The simplest way to start sending notifications is using channels. This allows you to use a publisher-subscriber model for sending pushes. Devices start by subscribing to one or more channels, and notifications can later be sent to these subscribers. The channels subscribed to by a given `Installation` are stored in the `channels` field of the `Installation` object.

A channel is identified by a string that starts with a letter and consists of alphanumeric characters, underscores, and dashes. It doesn't need to be explicitly created before it can be used and each `Installation` can subscribe to any number of channels at a time.

An installation's channels can be set using the `Channels` property of `ParseInstallation`. For example, in a baseball score app, we could do:

```
// When users indicate they are Giants fans, we
// subscribe them to that channel.
var installation =
ParseInstallation.CurrentInstallation;
installation.Channels = new List<string> {
    "Giants" };
installation.SaveAsync();
```

Alternatively, you can insert a channel into `Channels` without affecting the existing channels using the `AddUniqueToList` method of `ParseObject` using the following:

```
var installation =
ParseInstallation.CurrentInstallation;
installation.AddUniqueToList("channels",
    "Giants");
installation.SaveAsync();
```

Finally, `ParsePush` provides a shorthand for inserting a channel into `Channels` and saving:

```
ParsePush.SubscribeAsync("Giants");
```

Once subscribed to the “Giants” channel, your `Installation` object should have an updated `channels` field.

| objectId | String | deviceType | String | channels | Array | deviceToken | String | localeIdentifier | String |
|--------------------------|------------|------------|--------|------------|-------|------------------------|--------|------------------|--------|
| <input type="checkbox"/> | u7tvyZx1dJ | ios | | ["Giants"] | | 09c963ff7bc1c4775e6... | | en-US | (u |

Unsubscribing from a channel is just as easy:

```
var installation =
ParseInstallation.CurrentInstallation;
installation.RemoveAllFromList("channels" new
```

```
List<string> { "Giants" });  
installation.SaveAsync();
```

Or, using ParsePush:

```
ParsePush.UnsubscribeAsync("Giants");
```

The set of subscribed channels is cached in the `CurrentInstallation` object:

```
var installation =  
ParseInstallation.CurrentInstallation  
IEnumerable<string> subscribedChannels =  
installation.Channels;
```

If you plan on changing your channels from Cloud Code or the data browser, note that you'll need to call `FetchAsync` prior to this line in order to get the most recent channels.

SENDING PUSHES TO CHANNELS

In the .NET SDK, the following code can be used to alert all subscribers of the “Giants” channel that their favorite team just scored. This will display a toast notification to Windows users. iOS users will receive a notification in the notification center and Android users will receive a notification in the system tray.

```
// Send a notification to all devices subscribed  
to the "Giants" channel.  
var push = new ParsePush();  
push.Channels = new List<string> {"Giants"};  
push.Alert = "The Giants just scored!";  
push.SendAsync();
```

If you want to target multiple channels with a single push notification, you can use any `IEnumerable<string>` of channels.

USING ADVANCED TARGETING

While channels are great for many applications, sometimes you need more precision when targeting the recipients of your pushes. Parse allows you to write a query for any subset of your `Installation` objects using the [querying API](#) and to send them a push.

Since `ParseInstallation` is a subclass of `ParseObject`, you can save any data you want and even create relationships between `Installation` objects and your other objects. This allows you to send pushes to a very customized and dynamic segment of your user base.

SAVING INSTALLATION DATA

Storing data on an `Installation` object is just as easy as storing **any other data** on Parse. In our Baseball app, we could allow users to get pushes about game results, scores and injury reports.

```
// Store the category of push notifications the
user would like to receive.
var installation =
ParseInstallation.CurrentInstallation;
installation["scores"] = true;
installation["gameResults"] = true;
installation["injuryReports"] = true;
installation.SaveAsync();
```

You can even create relationships between your `Installation` objects and other classes saved on Parse. To associate an `Installation` with a particular user, for example, you can simply store the current user on the `ParseInstallation`.

```
// Associate the device with a user
var installation =
ParseInstallation.CurrentInstallation;
installation["user"] = ParseUser.CurrentUser;
installation.SaveAsync();
```

SENDING PUSHES TO QUERIES

Once you have your data stored on your `Installation` objects, you can use a `ParseQuery` to target a subset of these devices. `Installation` queries work just like any other **Parse query**, but we use the special static property `ParseInstallation.Query` to create it. We set this query on our `ParsePush` object, before sending the notification.

```
var push = new ParsePush();
push.Query = ParseInstallation.Query
    .WhereEqualTo("injuryReports",
true);
```

```
push.Alert = "Willie Hayes injured by own pop  
fly.";  
push.SendAsync();
```

We can even use channels with our query. To send a push to all subscribers of the “Giants” channel but filtered by those who want score update, we can do the following:

```
var push = new ParsePush();  
push.Query = ParseInstallation.Query  
    .WhereEqualTo("scores", true);  
push.Channels = new List<string> { "Giants" };  
push.Alert = "Giants scored against the A's! It's  
now 2-2.";  
push.SendAsync();
```

Alternatively, we can use a query that constrains “channels” directly:

```
var push = new ParsePush();  
push.Query = ParseInstallation.Query  
    .WhereEqualTo("scores", true)  
    .WhereContainsAll("channels", new  
string[] { "Giants" });  
push.Alert = "Giants scored against the A's! It's  
now 2-2.";  
push.SendAsync();
```

If we store relationships to other objects in our `Installation` class, we can also use those in our query. For example, we could send a push notification to all users near a given location like this.

```
// Find users in the Seattle metro area  
var userQuery =  
ParseUser.Query.WhereWithinDistance(  
    "location",  
    marinersStadium,  
    ParseGeoDistance.FromMiles(1));  
var push= new ParsePush();  
push.Query = ParseInstallation.Query  
    .WhereMatchesQuery("user",  
userQuery);  
push.Alert = "Mariners lost? Free conciliatory  
hotdogs at the Parse concession stand!";  
push.SendAsync();
```

Sending Options

Push notifications can do more than just send a message. On Unity, pushes can also include a title, as well as any custom data you wish to send. An expiration date can also be set for the notification in case it is time sensitive.

CUSTOMIZING YOUR NOTIFICATIONS

If you want to send more than just a message, you will need to use an `IDictionary<string, object>` to package all of the data. There are some reserved fields that have a special meaning.

- + `alert`: the notification's message.
- + `badge`: (*iOS only*) the value indicated in the top right corner of the app icon. This can be set to a value or to `Increment` in order to increment the current value by 1.
- + `sound`: (*iOS only*) the name of a sound file in the application bundle.
- + `content-available`: (*iOS only*) If you are a writing a **Newsstand** app, or an app using the Remote Notification Background Mode **introduced in iOS7** (a.k.a. "Background Push"), set this value to 1 to trigger a background download.
- + `category`: (*iOS only*) the identifier of the `UIApplicationCategory` for this push notification.
- + `uri`: (*Android only*) an optional field that contains a URI. When the notification is opened, an `Activity` associated with opening the URI is launched.
- + `title`: (*Android, Windows 8, and Windows Phone 8 only*) the value displayed in the Android system tray or Windows toast notification.

For example, to send a notification that contains a title, you can do the following:

```
var push = new ParsePush();
push.Channels = new List<string> {"Mets"};
push.Data = new Dictionary<string, object> {
    {"title", "Score Alert"}
    {"alert", "The Mets scored! The game is now tied 1-1!"},
```

```
};  
push.SendAsync();
```

SETTING AN EXPIRATION DATE

When a user's device is turned off or not connected to the internet, push notifications cannot be delivered. If you have a time sensitive notification that is not worth delivering late, you can set an expiration date. This avoids needlessly alerting users of information that may no longer be relevant.

There are two properties provided by the `ParsePush` class to allow setting an expiration date for your notification. The first is `Expiration` which simply takes a `DateTime?` specifying when Parse should stop trying to send the notification.

```
var push = new ParsePush();  
push.Expiration = new DateTime(2015, 8, 14);  
push.Alert = "Season tickets on sale until August  
14th!";  
push.SendAsync();
```

There is however a caveat with this method. Since device clocks are not guaranteed to be accurate, you may end up with inaccurate results. For this reason, the `ParsePush` class also provides the `ExpirationInterval` property which accepts a `TimeSpan`. The notification will expire after the specified interval has elapsed.

```
var push = new ParsePush();  
push.ExpirationInterval = TimeSpan.FromDays(7);  
push.Alert = "Season tickets on sale until next  
week!";  
push.SendAsync();
```

TARGETING BY PLATFORM

If you build a cross platform app, it is possible you may only want to target one operating system. There are two methods provided to filter which of these devices are targeted. Note that all platforms are targeted by default.

The following example would send a different notification to Android, iOS, and Windows users.

```
// Notification for Android users
var androidPush = new ParsePush();
androidPush.Query = ParseInstallation.Query
    .WhereContainsAll("channels",
new string[] { "suitcaseOwners" })
    .WhereEqualTo("deviceType",
"android");
androidPush.SendAsync();

// Notification for iOS users
+var iosPush = new ParsePush();
iosPush.Alert = "Your suitcase has been filled
with tiny apples!";
iosPush.Query = ParseInstallation.Query
    .WhereContainsAll("channels", new
string[] { "suitcaseOwners" })
    .WhereEqualTo("deviceType",
"ios");
iosPush.SendAsync();

// Notification for Windows 8 users
var winPush = new ParsePush();
winPush.Alert = "Your suitcase has been filled
with tiny glass!";
winPush.Query = ParseInstallation.Query
    .WhereContainsAll("channels", new
string[] { "suitcaseOwners" })
    .WhereEqualTo("deviceType",
"winrt");
winPush.SendAsync();

// Notification for Windows Phone 8 users
var wpPush = new ParsePush();
wpPush.Alert = "Your suitcase is very hip; very
metro.";
wpPush.Query = ParseInstallation.Query
    .WhereContainsAll("channels", new
string[] { "suitcaseOwners" })
    .WhereEqualTo("deviceType",
"winphone");
wpPush.SendAsync();
```

Scheduling Pushes

Sending scheduled push notifications is not currently supported by the .NET SDK. Take a look at the [REST API](#), [JavaScript SDK](#) or the Parse.com push console.

Receiving Pushes

If your app is running while a push notification is received, the

`ParsePush.ParsePushNotificationReceived` event is fired. You can register for this event. This event provides `ParsePushNotificationEventArgs`.

```
ParsePush.ParsePushNotificationReceived +=  
(sender, args) => {  
    var payload = args.Payload;  
    object objectId;  
    if (payload.TryGetValue("objectId", out  
objectId)) {  
        DisplayRichMessageWithObjectId(objectId as  
string);  
    }  
};
```

In Unity Android, we provide a helper method that you can utilize as a handler to display a notification. If the app is in background it will display a notification by default. `` `cs

```
ParsePush.ParsePushNotificationReceived += (sender,  
args) => { #if UNITY_ANDROID AndroidJavaClass  
parseUnityHelper = new  
AndroidJavaClass("com.parse.ParsePushUnityHelper");  
AndroidJavaClass unityPlayer = new  
AndroidJavaClass("com.unity3d.player.UnityPlayer");  
AndroidJavaObject currentActivity =  
unityPlayer.GetStatic("currentActivity");
```

```
// Call default behavior.  
parseUnityHelper.CallStatic("handleParsePushNotific  
currentActivity, args.StringPayload); #endif }  
``
```

TRACKING PUSHES AND APP OPENS

Tracking push opens is not supported on Unity now.
You can track app opens with

```
ParseAnalytics.TrackAppOpenedAsync();
```

Push Experiments

You can A/B test your push notifications to figure out the best way to keep your users engaged. With A/B testing, you can simultaneously send two versions of your push notification to different devices, and use each

version's push open rates to figure out which one is better. You can test by either message or send time.

A/B TESTING

Our web push console guides you through every step of setting up an A/B test.

For each push campaign sent through the Parse web push console, you can allocate a subset of your devices to be in the experiment's test audience, which Parse will automatically split into two equally-sized experiment groups. For each experiment group, you can specify a different push message. The remaining devices will be saved so that you can send the winning message to them later. Parse will randomly assign devices to each group to minimize the chance for a test to affect another test's results (although we still don't recommend running multiple A/B tests over the same devices on the same day).

A/B Testing

Experiment with different messages or send times to discover the optimal campaign variables.

Use A/B Testing

No ☒ Yes

Name your Test
Give your test a memorable name so you remember what you were testing when you see the results.

New Game Level Announcement

Test Variable
You can test the messaging or the delivery time.

message ☒ time

Test Size
Test with a subset to find the right messaging, and then send to the rest.

20%

Based on your campaign size, we recommend that you include all devices in the test audience for better results.

Your campaign will be tested with **11624** devices
(after your experiment is over, you'll be able to send the winner to the rest)

After you send the push, you can come back to the push console to see in real time which version resulted in more push opens, along with other metrics such as statistical confidence interval. It's normal for the number of recipients in each group to be slightly different because some devices that we had originally allocated to that experiment group may have uninstalled the app. It's also possible for the random group assignment to be slightly uneven when the test audience size is small. Since we calculate open rate separately for each group based on recipient count, this should not significantly affect your experiment results.



If you are happy with the way one message performed, you can send that to the rest of your app's devices (i.e. the "Launch Group"). This step only applies to A/B tests where you vary the message.



Push experiments are supported on all recent Parse SDKs (iOS v1.2.13+, Android v1.4.0+, .NET v1.2.7+). Before running experiments, you must instrument your app with [push open tracking](#).

EXPERIMENT STATISTICS

Parse provides guidance on how to run experiments to achieve statistically significant results.

TEST AUDIENCE SIZE

When you setup a push message experiment, we'll recommend the minimum size of your test audience. These recommendations are generated through simulations based on your app's historical push open rates. For big push campaigns (e.g. 100k+ devices), this recommendation is usually small subset of your devices. For smaller campaigns (e.g. < 5k devices), this recommendation is usually all devices. Using all devices for your test audience will not leave any remaining devices for the launch group, but you can still gain valuable insight into what type of messaging works better so you can implement similar messaging in your next push campaign.

CONFIDENCE INTERVAL

After you send your pushes to experiment groups, we'll also provide a statistical confidence interval when your experiment has collected enough data to have statistically significant results. This confidence interval is in absolute percentage points of push open rate (e.g. if the open rates for groups A and B are 3% and 5%, then the difference is reported as 2 percentage points). This confidence interval is a measure of how much

difference you would expect to see between the two groups if you repeat the same experiment many times.

Just after a push send, when only a small number of users have opened their push notifications, the open rate difference you see between groups A and B could be due to random chance, so it might not be reproducible if you run the same experiment again. After your experiment collects more data over time, we become increasingly confident that the observed difference is a true difference. As this happens, the confidence interval will become narrower, allowing us to more accurately estimate the true difference between groups A and B. Therefore, we recommend that you wait until there is enough data to generate a statistical confidence interval before deciding which group's push is better.

Push Localization

Localizing your app's content is a proven way to drive greater engagement. We've made it easy to localize your push messages with Push Localization. The latest version of the Parse .NET SDK will detect and store the user's language in the installation object, and via the web push console you'll be able to send localized push messages to your users in a single broadcast.

SETUP

To take advantage of Push Localization you will need to make sure you've published your app with the Parse .NET SDK version 1.5.5 or greater. Any users of your application running the Parse .NET SDK version 1.5.5 or greater will then be targetable by Push Localization via the web push console.

It's important to note that for developers who have users running apps with versions of the Parse .NET SDK earlier than 1.5.5 that targeting information for Localized Push will not be available and these users will receive the default message from the push console.

SENDING A LOCALIZED PUSH

Our web push console guides you through every step of setting up a Localized Push.

Troubleshooting

Setting up Push Notifications is often a source of frustration for developers. The process is complicated and invites problems to happen along the way. If you run into issues, try some of these troubleshooting tips.

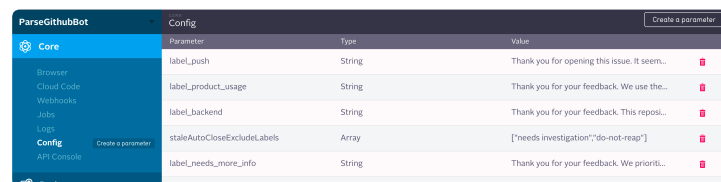
- + If you're receiving push from Unity iOS, refer to [iOS Push Troubleshooting Guide](#).
- + If you're receiving push from Unity Android, refer to [Android Push Troubleshooting Guide](#).

Want to contribute to this doc? [Edit this section](#).

Config

Parse Config

`ParseConfig` is a way to configure your applications remotely by storing a single configuration object on Parse. It enables you to add things like feature gating or a simple “Message of the Day”. To start using `ParseConfig` you need to add a few key/value pairs (parameters) to your app on the Parse Config Dashboard.



| Parameter | Type | Value |
|-----------------------------|--------|--|
| label_push | String | Thank you for opening this issue. It seem... |
| label_product_usage | String | Thank you for your feedback. We use the... |
| label_backend | String | Thank you for your feedback. This reposi... |
| staleAutoCloseExcludeLabels | Array | ["needs investigation","do-not-reap"] |
| label_needs_more_info | String | Thank you for your feedback. We priorit... |

After that you will be able to fetch the `ParseConfig` on the client, like in this example:

```
ParseConfig.GetAsync().ContinueWith(t =>
{
    if (t.IsFaulted) {
        // Something went wrong (e.g. request timed
out)
    } else {
        ParseConfig config = t.Result;
    }
})
```

Retrieving Config

`ParseConfig` is built to be as robust and reliable as possible, even in the face of poor internet connections.

Caching is used by default to ensure that the latest successfully fetched config is always available. In the below example we use `GetAsync` to retrieve the latest version of config from the server, and if the fetch fails we can simply fall back to the version that we successfully fetched before via `CurrentConfig`.

```
ParseConfig.GetAsync().ContinueWith(t =>
{
    ParseConfig config = null;
    if (t.IsFaulted) {
        Console.WriteLine("Failed to fetch. Using
Cached Config.");
        config = ParseConfig.CurrentConfig;
    } else {
        config = t.Result;
    }

    string welcomeMessage = null;
    bool result =
config.TryGetValue("welcomeMessage", out
welcomeMessage);
    if (!result) {
        Console.WriteLine("Falling back to default
message.");
        welcomeMessage = "Welcome!";
    }

    Console.WriteLine(String.Format("Welcome
Message From Config = {0}", welcomeMessage));
})
```

Current Config

Every `ParseConfig` instance that you get is always immutable. When you retrieve a new `ParseConfig` in the future from the network, it will not modify any existing `ParseConfig` instance, but will instead create a new one and make it available via `ParseConfig.CurrentConfig`. Therefore, you can safely pass around any `ParseConfig` object and safely assume that it will not automatically change.

It might be troublesome to retrieve the config from the server every time you want to use it. You can avoid this by simply using the cached `CurrentConfig` object and fetching the config only once in a while.

```

public class Helper
{
    private static TimeSpan configRefreshInterval =
TimeSpan.FromHours(12);
    private static DateTime? lastFetchedDate;

    // Fetches the config at most once every 12
hours per app runtime
    public static void FetchConfigIfNeeded()
    {
        if (lastFetchedDate == null ||
            DateTime.Now - lastFetchedDate >
configRefreshInterval) {
            lastFetchedDate = DateTime.Now;
            ParseConfig.GetAsync();
        }
    }
}

```

Parameters

`ParseConfig` supports most of the data types supported by `ParseObject`:

- + string
- + bool/int/double/long
- + DateTime
- + ParseFile
- + ParseGeoPoint
- + IList<string, T> (even nested)
- + IDictionary (even nested)

We currently allow up to **100** parameters in your config and a total size of **128KB** across all parameters.

Want to contribute to this doc? [Edit this section.](#)

Analytics

Parse provides a number of hooks for you to get a glimpse into the ticking heart of your app. We understand that it's important to understand what your app is doing, how frequently, and when.

While this section will cover different ways to instrument your app to best take advantage of Parse's analytics backend, developers using Parse to store and retrieve data can already take advantage of metrics on Parse.

Without having to implement any client-side logic, you can view real-time graphs and breakdowns (by device type, Parse class name, or REST verb) of your API Requests in your app's dashboard and save these graph filters to quickly access just the data you're interested in.

App-Open Analytics

Our initial analytics hook allows you to track your application being launched. By adding the following line to your Launching event handler, you'll be able to collect data on when and how often your application is opened.

```
ParseAnalytics.TrackAppOpenedAsync();
```

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

Custom Analytics

`ParseAnalytics` also allows you to track free-form events, with a handful of `string` keys and values. These extra dimensions allow segmentation of your custom events via your app's Dashboard.

Say your app offers search functionality for apartment listings, and you want to track how often the feature is used, with some additional metadata.

```
var dimensions = new Dictionary<string, string> {  
    // Define ranges to bucket data points into  
    meaningful segments  
    { "priceRange", "1000-1500" },  
    // Did the user filter the query?  
    { "source", "craigslist" },  
    // Do searches happen more often on weekdays or  
    weekends?  
    { "dayType", "weekday" }  
};  
// Send the dimensions to Parse along with the  
'search' event
```

```
ParseAnalytics.TrackEventAsync("search",  
dimensions);
```

`ParseAnalytics` can even be used as a lightweight error tracker — simply invoke the following and you'll have access to an overview of the rate and frequency of errors, broken down by error code, in your application:

```
var errDimensions = new Dictionary<string,  
string> {  
    { "code", Convert.ToString(error.Code) }  
};  
ParseAnalytics.TrackEventAsync("error",  
errDimensions );
```

Note that Parse currently only stores the first eight dimension pairs per call to

`ParseAnalytics.TrackEventAsync()`.

Want to contribute to this doc? [Edit this section.](#)

Data

We've designed the Parse SDKs so that you typically don't need to worry about how data is saved while using the client SDKs. Simply add data to the Parse `Object`, and it'll be saved correctly.

Nevertheless, there are some cases where it's useful to be aware of how data is stored on the Parse platform.

Data Storage

Internally, Parse stores data as JSON, so any datatype that can be converted to JSON can be stored on Parse. Refer to the [Data Types in Objects](#) section of this guide to see platform-specific examples.

Keys including the characters `$` or `.`, along with the key `__type` key, are reserved for the framework to handle additional types, so don't use those yourself. Key names must contain only numbers, letters, and underscore, and must start with a letter. Values can be anything that can be JSON-encoded.

Data Type Lock-in

When a class is initially created, it doesn't have an inherent schema defined. This means that for the first object, it could have any types of fields you want.

However, after a field has been set at least once, that field is locked into the particular type that was saved. For example, if a `User` object is saved with field `name` of type `String`, that field will be restricted to the `String` type only (the server will return an error if you try to save anything else).

One special case is that any field can be set to `null`, no matter what type it is.

The Data Browser

The Data Browser is the web UI where you can update and create objects in each of your apps. Here, you can see the raw JSON values that are saved that represents each object in your class.

When using the interface, keep in mind the following:

- + The `objectId`, `createdAt`, `updatedAt` fields cannot be edited (these are set automatically).
- + The value "(empty)" denotes that the field has not been set for that particular object (this is different than `null`).
- + You can remove a field's value by hitting your Delete key while the value is selected.

The Data Browser is also a great place to test the Cloud Code validations contained in your Cloud Code functions (such as `beforeSave`). These are run whenever a value is changed or object is deleted from the Data Browser, just as they would be if the value was changed or deleted from your client code.

Importing Data

You may import data into your Parse app by using CSV or JSON files. To create a new class with data from a CSV or JSON file, go to the Data Browser and click the "Import" button on the left hand column.

The JSON format is an array of objects in our REST format or a JSON object with a `results` that contains an array of objects. It must adhere to the [JSON standard](#). A file containing regular objects could look like:

```
{ "results": [  
  {  
    "score": 1337,  
    "playerName": "Sean Plott",  
    "cheatMode": false,  
    "createdAt": "2012-07-11T20:56:12.347Z",  
    "updatedAt": "2012-07-11T20:56:12.347Z",  
    "objectId": "fchpZwSuGG"  
  }  
]
```

Objects in either format should contain keys and values that also satisfy the following:

- + Key names must contain only numbers, letters, and underscore, and must start with a letter.
- + No value may contain a hard newline `'\n'`.

Normally, when objects are saved to Parse, they are automatically assigned a unique identifier through the `objectId` field, as well as a `createdAt` field and `updatedAt` field which represent the time that the object was created and last modified in the Parse Cloud. These fields can be manually set when data is imported from a JSON file. Please keep in mind the following:

- + Use a unique 10 character alphanumeric string as the value of your `objectId` fields.
- + Use a UTC timestamp in the ISO 8601 format when setting a value for the `createdAt` field or the `updatedAt` field.

In addition to the exposed fields, objects in the Parse User class can also have the `bcryptPassword` field set. The value of this field is a `String` that is the bcrypt hashed password + salt in the modular crypt format described in this [StackOverflow answer](#). Most OpenSSL based bcrypt implementations should have built-in methods to produce these strings.

A file containing a `User` object could look like:

```
{ "results":  
  [{  
    "username": "cooldude",  
    "createdAt": "1983-09-13T22:42:30.548Z",  
    "updatedAt": "2015-09-04T10:12:42.137Z",  
    "objectId": "ttttSEpfXm",  
    "sessionToken": "dfwfq3dh0zwe5y2sqv514p4ib",  
    "bcryptPassword":  
"$2a$10$ICV5UeEf3lICfnE9W9pN9.09Ved/ozNo7G83Qbdk5rr"  
  }]  
}
```

Note that in CSV the import field types are limited to `String`, `Boolean`, and `Number`.

Exporting your Data

You can request an export of your data at any time from your app's Settings page. The data export runs at a lower priority than production queries, so if your app is still serving queries, production traffic will always be given a higher priority, which may slow down the delivery of your data export.

EXPORT FORMATS

Each collection will be exported in the same JSON format used by our REST API and delivered in a single zipped file. Since data is stored internally as JSON, this allows us to ensure that the export closely matches how the data is saved to Parse. Other formats such as CSV cannot represent all of the data types supported by Parse without losing information. If you'd like to work with your data in CSV format, you can use any of the JSON-to-CSV converters available widely on the web.

OFFLINE ANALYSIS

For offline analysis of your data, we highly recommend using alternate ways to access your data that do not require extracting the entire collection at once. For example, you can try exporting only the data that has changed since your last export. Here are some ways of achieving this:

- + Use the JavaScript SDK in a node app.
`Parse.Query.each()` will allow you to extract every single object that matches a query. You can use date constraints to make sure the query only matches data that has been updated since you last ran this app. Your node app can write this data to disk for offline analysis.
- + Use the REST API in a script. You can run queries against your class and use `skip/limit` to page through results, which can then be written to disk for offline analysis. You can again use date constraints to make sure only newly updated data is extracted.
- + If the above two options do not fit your needs, you can try using the Data Browser to export data selectively. Use the Funnel icon to create a filter for the specific data that you need to export, such as newly updated objects. Once the filter has been applied, click on the Export data icon on the upper right of your Data Browser. This type of export will only include the objects that match your criteria.

Want to contribute to this doc? [Edit this section.](#)

Relations

There are three kinds of relationships. One-to-one relationships enable one object to be associated with another object. One-to-many relationships enable one object to have many related objects. Finally, many-to-many relationships enable complex relationships among many objects.

There are four ways to build relationships in Parse:

One-to-Many

When you're thinking about one-to-many relationships and whether to implement Pointers or Arrays, there are several factors to consider. First, how many objects are involved in this relationship? If the "many" side of the relationship could contain a very large number (greater

than 100 or so) of objects, then you have to use Pointers. If the number of objects is small (fewer than 100 or so), then Arrays may be more convenient, especially if you typically need to get all of the related objects (the “many” in the “one-to-many relationship”) at the same time as the parent object.

USING POINTERS

Let’s say we have a game app. The game keeps track of the player’s score and achievements every time she chooses to play. In Parse, we can store this data in a single `Game` object. If the game becomes incredibly successful, each player will store thousands of `Game` objects in the system. For circumstances like this, where the number of relationships can be arbitrarily large, Pointers are the best option.

Suppose in this game app, we want to make sure that every `Game` object is associated with a Parse User. We can implement this like so:

```
var game = new ParseObject("Game");
game["createdBy"] = ParseUser.CurrentUser;
```

We can obtain all of the `Game` objects created by a Parse User with a query:

```
var query =
ParseObject.getQuery("Game").WhereEqualTo("createdBy",
ParseUser.CurrentUser);
```

And, if we want to find the Parse User who created a specific `Game`, that is a lookup on the `createdBy` key:

```
// say we have a Game object
ParseObject game = ...

// getting the user who created the Game
ParseUser user = game["createdBy"];
```

For most scenarios, Pointers will be your best bet for implementing one-to-many relationships.

USING ARRAYS

Arrays are ideal when we know that the number of objects involved in our one-to-many relationship are

going to be small. Arrays will also provide some productivity benefit via the `includeKey` parameter. Supplying the parameter will enable you to obtain all of the “many” objects in the “one-to-many” relationship at the same time that you obtain the “one” object. However, the response time will be slower if the number of objects involved in the relationship turns out to be large.

Suppose in our game, we enabled players to keep track of all the weapons their character has accumulated as they play, and there can only be a dozen or so weapons. In this example, we know that the number of weapons is not going to be very large. We also want to enable the player to specify the order in which the weapons will appear on screen. Arrays are ideal here because the size of the array is going to be small and because we also want to preserve the order the user has set each time they play the game:

Let's start by creating a column on our Parse User object called `weaponsList`.

Now let's store some `Weapon` objects in the `weaponsList`:

```
// let's say we have four weapons
var scimitar = ...
var plasmaRifle = ...
var grenade = ...
var bunnyRabbit = ...

// stick the objects in an array
var weapons = new List<ParseObject>();
weapons.Add(scimitar);
weapons.Add(plasmaRifle);
weapons.Add(grenade);
weapons.Add(bunnyRabbit);

// store the weapons for the user
var user = ParseUser.CurrentUser;
user.AddRangeToList("weaponsList", weapons);
```

Later, if we want to retrieve the `Weapon` objects, it's just one line of code:

```
var weapons =
ParseUser.CurrentUser.Get<IList<Object>>
("weaponsList");
```

Sometimes, we will want to fetch the “many” objects in our one-to-many relationship at the same time as we fetch the “one” object. One trick we could employ is to use the `includeKey` (or `include` in Android) parameter whenever we use a Parse Query to also fetch the array of `Weapon` objects (stored in the `weaponsList` column) along with the Parse User object:

```
// set up our query for a User object
var userQuery = ParseUser.Query;

// configure any constraints on your query...
// for example, you may want users who are also
// playing with or against you

// tell the query to fetch all of the Weapon
// objects along with the user
// get the "many" at the same time that you're
// getting the "one"
userQuery = userQuery.Include("weaponsList");

// execute the query
IEnumerable<ParseUser> results = await
userQuery.FindAsync();
// results contains all of the User objects, and
// their associated Weapon objects, too
```

You can also get the “one” side of the one-to-many relationship from the “many” side. For example, if we want to find all Parse User objects who also have a given `Weapon`, we can write a constraint for our query like this:

```
// add a constraint to query for whenever a
// specific Weapon is in an array
userQuery = userQuery.WhereEqualTo("weaponsList",
scimitar);

// or query using an array of Weapon objects...
userQuery =
userQuery.WhereContainedIn("weaponsList",
arrayOfWeapons);
```

Many-to-Many

Now let’s tackle many-to-many relationships. Suppose we had a book reading app and we wanted to model `Book` objects and `Author` objects. As we know, a given author can write many books, and a given book can have multiple authors. This is a many-to-many relationship scenario where you have to choose between Arrays, Parse Relations, or creating your own Join Table.

The decision point here is whether you want to attach any metadata to the relationship between two entities. If you don't, Parse Relation or using Arrays are going to be the easiest alternatives. In general, using arrays will lead to higher performance and require fewer queries. If either side of the many-to-many relationship could lead to an array with more than 100 or so objects, then, for the same reason Pointers were better for one-to-many relationships, Parse Relation or Join Tables will be better alternatives.

On the other hand, if you want to attach metadata to the relationship, then create a separate table (the "Join Table") to house both ends of the relationship. Remember, this is information **about the relationship**, not about the objects on either side of the relationship. Some examples of metadata you may be interested in, which would necessitate a Join Table approach, include:

USING PARSE RELATIONS

Using Parse Relations, we can create a relationship between a `Book` and a few `Author` objects. In the Data Browser, you can create a column on the `Book` object of type relation and name it `authors`.

After that, we can associate a few authors with this book:

```
// let's say we have a few objects representing
// Author objects
var authorOne = ...
var authorTwo = ...
var authorThree = ...

// now we create a book object
var book = new ParseObject("Book");

// now let's associate the authors with the book
// remember, we created a "authors" relation on
// Book
var relation = book.GetRelation<ParseObject>
("authors");
relation.Add(authorOne);
relation.Add(authorTwo);
relation.Add(authorThree);

// now save the book object
await book.SaveAsync();
```

To get the list of authors who wrote a book, create a query:

```
// suppose we have a book object
var book = ...

// create a relation based on the authors key
var relation = book.GetRelation<ParseObject>
("authors");

// generate a query based on that relation
var query = relation.Query;

// now execute the query
```

Perhaps you even want to get a list of all the books to which an author contributed. You can create a slightly different kind of query to get the inverse of the relationship:

```
// suppose we have a author object, for which we
want to get all books
var author = ...

// first we will create a query on the Book
object
var query = ParseObject.GetQuery("Book");

// now we will query the authors relation to see
if the author object we have
// is contained therein
query = query.WhereEqualTo("authors", author);
```

USING JOIN TABLES

There may be certain cases where we want to know more about a relationship. For example, suppose we were modeling a following/follower relationship between users: a given user can follow another user, much as they would in popular social networks. In our app, we not only want to know if User A is following User B, but we also want to know **when** User A started following User B. This information could not be contained in a Parse Relation. In order to keep track of this data, you must create a separate table in which the relationship is tracked. This table, which we will call **Follow**, would have a **from** column and a **to** column, each with a pointer to a Parse User. Alongside the relationship, you can also add a column with a **Date** object named **date**.

Now, when you want to save the following relationship between two users, create a row in the **Follow** table, filling in the **from**, **to**, and **date** keys appropriately:

```
// suppose we have a user we want to follow
ParseUser otherUser = ...
```

```
// create an entry in the Follow table
var follow = new ParseObject("Follow");
follow["from"] = ParseUser.CurrentUser;
follow["to"] = otherUser;
follow["date"] = DateTime.UtcNow;
await follow.SaveAsync();
```

If we want to find all of the people we are following, we can execute a query on the `Follow` table:

```
// set up the query on the Follow table
ParseQuery<ParseObject> query =
ParseQuery.GetQuery("Follow");
query = query.WhereEqualTo("from",
ParseUser.CurrentUser);

// execute the query
IEnumerable<ParseObject> results = await
query.FindAsync();
```

It's also pretty easy to find all the users that are following the current user by querying on the `to` key:

```
// create an entry in the Follow table
var query = ParseObject.GetQuery("Follow")
    .WhereEqualTo("to", ParseUser.CurrentUser);
IEnumerable<ParseObject> results = await
query.FindAsync();
```

USING AN ARRAY

Arrays are used in Many-to-Many relationships in much the same way that they are for One-to-Many relationships. All objects on one side of the relationship will have an Array column containing several objects on the other side of the relationship.

Suppose we have a book reading app with `Book` and `Author` objects. The `Book` object will contain an Array of `Author` objects (with a key named `authors`). Arrays are a great fit for this scenario because it's highly unlikely that a book will have more than 100 or so authors. We will put the Array in the `Book` object for this reason. After all, an author could write more than 100 books.

Here is how we save a relationship between a `Book` and an `Author`.


```
// let's say we have an author
var author = ...

// and let's also say we have a book
var book = ...

// add the author to the authors list for the book
book.AddToList("authors", author);
```

Because the author list is an Array, you should use the `includeKey` (or `include` on Android) parameter when fetching a `Book` so that Parse returns all the authors when it also returns the book:

```
// set up our query for the Book object
var bookQuery = ParseObject.GetQuery("Book");

// configure any constraints on your query...
// tell the query to fetch all of the Author
// objects along with the Book
bookQuery = bookQuery.Include("authors");

// execute the query
IEnumerable<ParseObject> books= await
bookQuery.FindAsync();
```

At that point, getting all the `Author` objects in a given `Book` is a pretty straightforward call:

```
var authorList = book.Get<List<ParseObject>>
("authors");
```

Finally, suppose you have an `Author` and you want to find all the `Book` objects in which she appears. This is also a pretty straightforward query with an associated constraint:

```
// set up our query for the Book object
var bookQuery = ParseObject.GetQuery("Book");

// configure any constraints on your query...
bookQuery = bookQuery.WhereEqualTo("authors",
author);

// tell the query to fetch all of the Author
// objects along with the Book
bookQuery = bookQuery.Include("authors");

// execute the query
IEnumerable<ParseObject> books = await
bookQuery.FindAsync();
```

One-to-One

In Parse, a one-to-one relationship is great for situations where you need to split one object into two objects. These situations should be rare, but two examples include:

- + **Limiting visibility of some user data.** In this scenario, you would split the object in two, where one portion of the object contains data that is visible to other users, while the related object contains data that is private to the original user (and protected via ACLs).
- + **Splitting up an object for size.** In this scenario, your original object is greater than the 128K maximum size permitted for an object, so you decide to create a secondary object to house extra data. It is usually better to design your data model to avoid objects this large, rather than splitting them up. If you can't avoid doing so, you can also consider storing large data in a Parse File.

Thank you for reading this far. We apologize for the complexity. Modeling relationships in data is a hard subject, in general. But look on the bright side: it's still easier than relationships with people.

Want to contribute to this doc? [Edit this section.](#)

Handling Errors

Parse has a few simple patterns for surfacing errors and handling them in your code.

There are two types of errors you may encounter. The first is those dealing with logic errors in the way you're using the SDK. These types of errors result in general `Exception` being raised. For an example take a look at the following code:

```
var user = new ParseUser();
user.SignUpAsync();
```

This will throw an `InvalidOperationException` because `SignUpAsync` was called without first setting the required properties (`Username` and `Password`).

The second type of error is one that occurs when interacting with the Parse Cloud over the network. These errors are either related to problems connecting to the cloud or problems performing the requested operation. For example, an error may result from an existing user trying to sign up. Here's how you could handle this error as well as the previous SDK logic errors:

```
try
{
    user.SignUpAsync().ContinueWith(t => {
        if (t.IsFaulted) {
            // Errors from Parse Cloud and
network interactions
            using (IEnumerator<System.Exception>
enumerator =
t.Exception.InnerExceptions.GetEnumerator()) {
                if (enumerator.MoveNext()) {
                    ParseException error =
(ParseException) enumerator.Current;
                    // error.Message will contain
an error message
                    // error.Code will return
"OtherCause"
                }
            }
        }
    });
}
catch (InvalidOperationException e)
{
    // Error from the SDK logic checks
    // e.Message will contain the specific error
    // ex: "Cannot sign up user with an empty
name."
}
```

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

- 1 Due to limitations with Unity's `WWW` class, error details from Parse Cloud interactions are not passed back to the SDK. The error message in these scenarios is of the form "40x message" for example, "400 Bad Request" or "404 Not Found". You can implement a generic error handler for those scenarios.
- 2 If you want finer grained control over your error handling, you could look into using [Cloud Functions](#) to wrap your calls and send back the error information via the Cloud Code success path. Sending the error information through the

success path ensures that they are passed back to the SDK.

Let's take a look at another error handling example:

```
ParseObject.GetQuery("Note").GetAsync("thisObjectI
```

In the above code, we try to fetch an object with a non-existent `ObjectId`. The Parse Cloud will return an error – so here's how to handle it properly:

```
ParseObject.GetQuery("Note").GetAsync(someObjectId
=>
{
    if (t.IsFaulted)
    {
        // One or more errors occurred.
    }
    else
    {
        // Everything went fine!
    }
})
```

For a list of all possible `ErrorCode` types, scroll down to [Error Codes](#), or see the `ParseException.ErrorCode` section of the [.NET API](#).

Want to contribute to this doc? [Edit this section](#).

Security

As your app development progresses, you will want to use Parse's security features in order to safeguard data. This document explains the ways in which you can secure your apps.

If your app is compromised, it's not only you as the developer who suffers, but potentially the users of your app as well. Continue reading for our suggestions for sensible defaults and precautions to take before releasing your app into the wild.

Client vs. Server

When an app first connects to Parse, it identifies itself with an Application ID and a Client key (or REST Key, or .NET Key, or JavaScript Key, depending on which platform you're using). These are not secret and by themselves they do not secure an app. These keys are shipped as a part of your app, and anyone can decompile your app or proxy network traffic from their device to find your client key. This exploit is even easier with JavaScript — one can simply “view source” in the browser and immediately find your client key.

This is why Parse has many other security features to help you secure your data. The client key is given out to your users, so anything that can be done with just the client key is doable by the general public, even malicious hackers.

The master key, on the other hand, is definitely a security mechanism. Using the master key allows you to bypass all of your app's security mechanisms, such as [class-level permissions](#) and [ACLs](#). Having the master key is like having root access to your app's servers, and you should guard your master key with the same zeal with which you would guard your production machines' root password.

The overall philosophy is to limit the power of your clients (using client keys), and to perform any sensitive actions requiring the master key in Cloud Code. You'll learn how to best wield this power in the section titled [Implementing Business Logic in Cloud Code](#).

A final note: All connections are made with HTTPS and SSL, and Parse will reject all non-HTTPS connections. As a result, you don't need to worry about man-in-the-middle attacks.

Class-Level Permissions

The second level of security is at the schema and data level. Enforcing security measures at this level will restrict how and when client applications can access and create data on Parse. When you first begin developing your Parse application, all of the defaults are set so that you can be a more productive developer. For example:

- + A client application can create new classes on Parse
- + A client application can add fields to classes
- + A client application can modify or query for objects on Parse

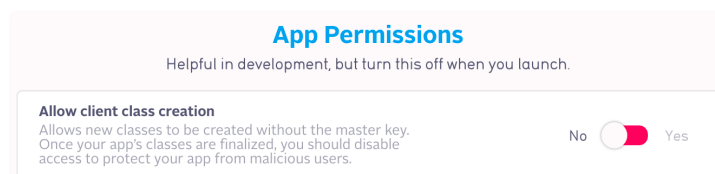
You can configure any of these permissions to apply to everyone, no one, or to specific users or roles in your app. Roles are groups that contain users or other roles, which you can assign to an object to restrict its use. Any permission granted to a role is also granted to any of its children, whether they are users or other roles, enabling you to create an access hierarchy for your apps. Each of [the Parse guides](#) includes a detailed description of employing Roles in your apps.

Once you are confident that you have the right classes and relationships between classes in your app, you should begin to lock it down by doing the following:

Almost every class that you create should have these permissions tweaked to some degree. For classes where every object has the same permissions, class-level settings will be most effective. For example, one common use case entails having a class of static data that can be read by anyone but written by no one.

RESTRICTING CLASS CREATION

As a start, you can configure your application so that clients cannot create new classes on Parse. This is done from the Settings tab on the Data Browser. Scroll down to the **App Permissions** section and turn off **Allow client class creation**. Once enabled, classes may only be created from the Data Browser. This will prevent attackers from filling your database with unlimited, arbitrary new classes.



CONFIGURING CLASS-LEVEL PERMISSIONS

Parse lets you specify what operations are allowed per class. This lets you restrict the ways in which clients can access or modify your classes. To change these settings, go to the Data Browser, select a class, and click the “Security” button.

You can configure the client's ability to perform each of the following operations for the selected class:

+ **Read:**

- + **Get:** With Get permission, users can fetch objects in this table if they know their objectIds.
- + **Find:** Anyone with Find permission can query all of the objects in the table, even if they don't know their objectIds. Any table with public Find permission will be completely readable by the public, unless you put an ACL on each object.

+ **Write:**

- + **Update:** Anyone with Update permission can modify the fields of any object in the table that doesn't have an ACL. For publicly readable data, such as game levels or assets, you should disable this permission.
 - + **Create:** Like Update, anyone with Create permission can create new objects of a class. As with the Update permission, you'll probably want to turn this off for publicly readable data.
 - + **Delete:** With this permission, people can delete any object in the table that doesn't have an ACL. All they need is its objectId.
- + **Add fields:** Parse classes have schemas that are inferred when objects are created. While you're developing your app, this is great, because you can add a new field to your object without having to make any changes on the backend. But once you ship your app, it's very rare to need to add new fields to your classes automatically. You should pretty much always turn off this permission for all of your classes when you submit your app to the public.

For each of the above actions, you can grant permission to all users (which is the default), or lock permissions

down to a list of roles and users. For example, a class that should be available to all users would be set to read-only by only enabling get and find. A logging class could be set to write-only by only allowing creates. You could enable moderation of user-generated content by providing update and delete access to a particular set of users or roles.

Object-Level Access Control

Once you've locked down your schema and class-level permissions, it's time to think about how data is accessed by your users. Object-level access control enables one user's data to be kept separate from another's, because sometimes different objects in a class need to be accessible by different people. For example, a user's private personal data should be accessible only to them.

Parse also supports the notion of anonymous users for those apps that want to store and protect user-specific data without requiring explicit login.

When a user logs into an app, they initiate a session with Parse. Through this session they can add and modify their own data but are prevented from modifying other users' data.

ACCESS CONTROL LISTS

The easiest way to control who can access which data is through access control lists, commonly known as ACLs. The idea behind an ACL is that each object has a list of users and roles along with what permissions that user or role has. A user needs read permissions (or must belong to a role that has read permissions) in order to retrieve an object's data, and a user needs write permissions (or must belong to a role that has write permissions) in order to update or delete that object.

Once you have a User, you can start using ACLs. Remember: Users can be created through traditional username/password signup, through a third-party login system like Facebook or Twitter, or even by using Parse's **automatic anonymous users** functionality. To set an ACL on the current user's data to not be publicly readable, all you have to do is:


```
var user = ParseUser.CurrentUser;  
user.ACL = new ParseACL(user);
```

Most apps should do this. If you store any sensitive user data, such as email addresses or phone numbers, you need to set an ACL like this so that the user's private information isn't visible to other users. If an object doesn't have an ACL, it's readable and writeable by everyone. The only exception is the `_User` class. We never allow users to write each other's data, but they can read it by default. (If you as the developer need to update other `_User` objects, remember that your master key can provide the power to do this.)

To make it super easy to create user-private ACLs for every object, we have a way to set a default ACL that will be used for every new object you create:

```
// not available in the .NET SDK
```

If you want the user to have some data that is public and some that is private, it's best to have two separate objects. You can add a pointer to the private data from the public one.

```
var privateData = new  
ParseObject("PrivateUserData");  
privateData.ACL = new  
ParseACL(ParseUser.CurrentUser);  
privateData["phoneNumber"] = "555-5309";  
  
ParseUser.CurrentUser["privateData"] =  
privateData;
```

Of course, you can set different read and write permissions on an object. For example, this is how you would create an ACL for a public post by a user, where anyone can read it:

```
var acl = new ParseACL();  
acl.PublicReadAccess = true;  
acl.SetRoleWriteAccess(ParseUser.CurrentUser.Object  
true);
```

Sometimes it's inconvenient to manage permissions on a per-user basis, and you want to have groups of users

who get treated the same (like a set of admins with special powers). Roles are a special kind of object that let you create a group of users that can all be assigned to the ACL. The best thing about roles is that you can add and remove users from a role without having to update every single object that is restricted to that role. To create an object that is writeable only by admins:

```
var acl = new ParseACL();
acl.PublicReadAccess = true;
acl.SetRoleWriteAccess("admins", true);
```

Of course, this snippet assumes you’ve already created a role named “admins”. This is often reasonable when you have a small set of special roles set up while developing your app. Roles can also be created and updated on the fly — for example, adding new friends to a “friendOf___” role after each connection is made.

All this is just the beginning. Applications can enforce all sorts of complex access patterns through ACLs and class-level permissions. For example:

- + For private data, read and write access can be restricted to the owner.
- + For a post on a message board, the author and members of the “Moderators” role can have “write” access, and the general public can have “read” access.
- + For logging data that will only be accessed by the developer through the REST API using the master key, the ACL can deny all permissions.
- + Data created by a privileged group of users or the developer, like a global message of the day, can have public read access but restrict write access to an “Administrators” role.
- + A message sent from one user to another can give “read” and “write” access just to those users.

For the curious, here’s the format for an ACL that restricts read and write permissions to the owner (whose `objectId` is identified by `"aSaMpLeUsErId"`) and enables other users to read the object:

```
{
  "*": { "read":true },
  "aSaMpLeUsErId": { "read" :true, "write":
```

```
true }  
}
```

And here's another example of the format of an ACL that uses a Role:

```
{  
  "role:RoleName": { "read": true },  
  "aSaMPLeUsErId": { "read": true, "write":  
true }  
}
```

POINTER PERMISSIONS

Pointer permissions are a special type of class-level permission that create a virtual ACL on every object in a class, based on users stored in pointer fields on those objects. For example, given a class with an `owner` field, setting a read pointer permission on `owner` will make each object in the class only readable by the user in that object's `owner` field. For a class with a `sender` and a `reciever` field, a read pointer permission on the `receiver` field and a read and write pointer permission on the `sender` field will make each object in the class readable by the user in the `sender` and `receiver` field, and writable only by the user in the `sender` field.

Given that objects often already have pointers to the user(s) that should have permissions on the object, pointer permissions provide a simple and fast solution for securing your app using data which is already there, that doesn't require writing any client code or cloud code.

Pointer permissions are like virtual ACLs. They don't appear in the ACL column, but if you are familiar with how ACLs work, you can think of them like ACLs. In the above example with the `sender` and `receiver`, each object will act as if it has an ACL of:

```
{  
  "<SENDER_USER_ID>": {  
    "read": true,  
    "write": true  
  },  
  "<RECEIVER_USER_ID>": {  
    "read": true  
  }  
}
```

```
}  
}  
}
```

Note that this ACL is not actually created on each object. Any existing ACLs will not be modified when you add or remove pointer permissions, and any user attempting to interact with an object can only interact with the object if both the virtual ACL created by the pointer permissions, and the real ACL already on the object allow the interaction. For this reason, it can sometimes be confusing to combine pointer permissions and ACLs, so we recommend using pointer permissions for classes that don't have many ACLs set. Fortunately, it's easy to remove pointer permissions if you later decide to use Cloud Code or ACLs to secure your app.

```
REQUIRES AUTHENTICATION PERMISSION  
(REQUIRES PARSE-SERVER >= 2.3.0)
```

Starting version 2.3.0, parse-server introduces a new Class Level Permission `requiresAuthentication`. This CLP prevents any non authenticated user from performing the action protected by the CLP.

For example, you want to allow your **authenticated users** to `find` and `get` `Announcement`'s from your application and your **admin role** to have all privileged, you would set the CLP:

```
// POST http://my-parse-  
server.com/schemas/Announcement  
// Set the X-Parse-Application-Id and X-Parse-  
Master-Key header  
// body:  
{  
  classLevelPermissions:  
  {  
    "find": {  
      "requiresAuthentication": true,  
      "role:admin": true  
    },  
    "get": {  
      "requiresAuthentication": true,  
      "role:admin": true  
    },  
    "create": { "role:admin": true },  
    "update": { "role:admin": true },  
    "delete": { "role:admin": true },  
  }  
}
```

Effects:

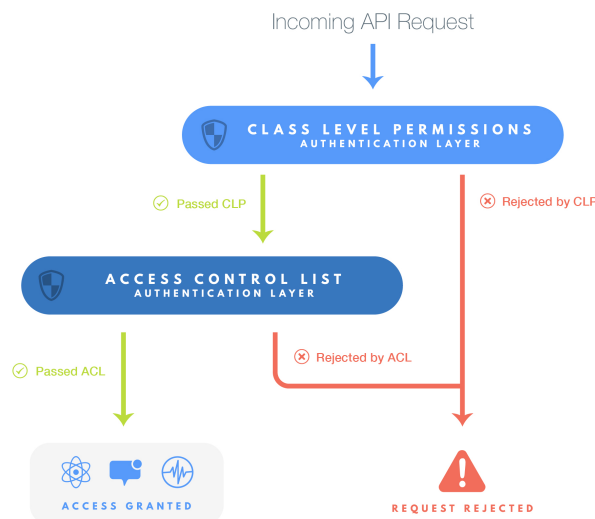
- + Non authenticated users won't be able to do anything.

- + Authenticated users (any user with a valid sessionToken) will be able to read all the objects in that class
- + Users belonging to the admin role, will be able to perform all operations.

:warning: Note that this is in no way securing your content, if you allow anyone to login to your server, every client will still be able to query this object.

CLP AND ACL INTERACTION

Class-Level Permissions (CLPs) and Access Control Lists (ACLs) are both powerful tools for securing your app, but they don't always interact exactly how you might expect. They actually represent two separate layers of security that each request has to pass through to return the correct information or make the intended change. These layers, one at the class level, and one at the object level, are shown below. A request must pass through BOTH layers of checks in order to be authorized. Note that despite acting similarly to ACLs, **Pointer Permissions** are a type of class level permission, so a request must pass the pointer permission check in order to pass the CLP check.



As you can see, whether a user is authorized to make a request can become complicated when you use both CLPs and ACLs. Let's look at an example to get a better sense of how CLPs and ACLs can interact. Say we have a `Photo` class, with an object, `photoObject`. There are 2 users in our app, `user1` and `user2`. Now let's say we set a Get CLP on the `Photo` class, disabling public Get, but allowing `user1` to perform Get. Now let's also set

an ACL on `photoObject` to allow Read - which includes GET - for only `user2`.

You may expect this will allow both `user1` and `user2` to Get `photoObject`, but because the CLP layer of authentication and the ACL layer are both in effect at all times, it actually makes it so *neither* `user1` nor `user2` can Get `photoObject`. If `user1` tries to Get `photoObject`, it will get through the CLP layer of authentication, but then will be rejected because it does not pass the ACL layer. In the same way, if `user2` tries to Get `photoObject`, it will also be rejected at the CLP layer of authentication.

Now let's look at an example that uses Pointer Permissions. Say we have a `Post` class, with an object, `myPost`. There are 2 users in our app, `poster`, and `viewer`. Let's say we add a pointer permission that gives anyone in the `Creator` field of the `Post` class read and write access to the object, and for the `myPost` object, `poster` is the user in that field. There is also an ACL on the object that gives read access to `viewer`. You may expect that this will allow `poster` to read and edit `myPost`, and `viewer` to read it, but `viewer` will be rejected by the Pointer Permission, and `poster` will be rejected by the ACL, so again, neither user will be able to access the object.

Because of the complex interaction between CLPs, Pointer Permissions, and ACLs, we recommend being careful when using them together. Often it can be useful to use CLPs only to disable all permissions for a certain request type, and then using Pointer Permissions or ACLs for other request types. For example, you may want to disable Delete for a `Photo` class, but then put a Pointer Permission on `Photo` so the user who created it can edit it, just not delete it. Because of the especially complex way that Pointer Permissions and ACLs interact, we usually recommend only using one of those two types of security mechanisms.

SECURITY EDGE CASES

There are some special classes in Parse that don't follow all of the same security rules as every other class. Not all classes follow **Class-Level Permissions (CLPs)** or **Access Control Lists (ACLs)** exactly how they are defined, and here those exceptions are

documented. Here “normal behavior” refers to CLPs and ACLs working normally, while any other special behaviors are described in the footnotes.

| | _User | _Installation |
|--------------|-------------------------------|---------------------------------|
| Get | normal behaviour
[1, 2, 3] | ignores CLP, but not
ACL |
| Find | normal behavior
[3] | master key only [6] |
| Create | normal behavior
[4] | ignores CLP |
| Update | normal behavior
[5] | ignores CLP, but not
ACL [7] |
| Delete | normal behavior
[5] | master key only [7] |
| Add
Field | normal behavior | normal behavior |

- 1 Logging in, or `/1/login` in the REST API, does not respect the Get CLP on the user class. Login works just based on username and password, and cannot be disabled using CLPs.
- 2 Retrieving the current user, or becoming a User based on a session token, which are both `/1/users/me` in the REST API, do not respect the Get CLP on the user class.
- 3 Read ACLs do not apply to the logged in user. For example, if all users have ACLs with Read disabled, then doing a find query over users will still return the logged in user. However, if the Find CLP is disabled, then trying to perform a find on users will still return an error.
- 4 Create CLPs also apply to signing up. So disabling Create CLPs on the user class also disables people from signing up without the master key.
- 5 Users can only Update and Delete themselves. Public CLPs for Update and Delete may still apply. For example, if you disable public Update for the user class, then users cannot edit themselves. But no matter what the write ACL on a user is, that user can still Update or Delete itself, and no other user can Update or Delete that user. As always, however, using the master key allows users to update other users, independent of CLPs or ACLs.

- 6 Get requests on installations follow ACLs normally. Find requests without master key is not allowed unless you supply the `installationId` as a constraint.
- 7 Update requests on installations do adhere to the ACL defined on the installation, but Delete requests are master-key-only. For more information about how installations work, check out the [installations section of the REST guide](#).

Data Integrity in Cloud Code

For most apps, care around keys, class-level permissions, and object-level ACLs are all you need to keep your app and your users' data safe. Sometimes, though, you'll run into an edge case where they aren't quite enough. For everything else, there's **Cloud Code**.

Cloud Code allows you to upload JavaScript to Parse's servers, where we will run it for you. Unlike client code running on users' devices that may have been tampered with, Cloud Code is guaranteed to be the code that you've written, so it can be trusted with more responsibility.

One particularly common use case for Cloud Code is preventing invalid data from being stored. For this sort of situation, it's particularly important that a malicious client not be able to bypass the validation logic.

To create validation functions, Cloud Code allows you to implement a `beforeSave` trigger for your class. These triggers are run whenever an object is saved, and allow you to modify the object or completely reject a save. For example, this is how you create a **Cloud Code `beforeSave` trigger** to make sure every user has an email address set:

```
Parse.Cloud.beforeSave(Parse.User,
function(request, response) {
  var user = request.object;
  if (!user.get("email")) {
    response.error("Every user must have an email address.");
  } else {
    response.success();
  }
});
```


Our [Cloud Code guide](#) provides instructions on how to upload this trigger to our servers.

Validations can lock down your app so that only certain values are acceptable. You can also use `afterSave` validations to normalize your data (e.g. formatting all phone numbers or currency identically). You get to retain most of the productivity benefits of accessing Parse data directly from your client applications, but you can also enforce certain invariants for your data on the fly.

Common scenarios that warrant validation include:

- + Making sure phone numbers have the right format
- + Sanitizing data so that its format is normalized
- + Making sure that an email address looks like a real email address
- + Requiring that every user specifies an age within a particular range
- + Not letting users directly change a calculated field
- + Not letting users delete specific objects unless certain conditions are met

Implementing Business Logic in Cloud Code

While validation often makes sense in Cloud Code, there are likely certain actions that are particularly sensitive, and should be as carefully guarded as possible. In these cases, you can remove permissions or the logic from clients entirely and instead funnel all such operations to Cloud Code functions.

When a Cloud Code function is called, it can invoke the `useMasterKey` function to gain the ability to modify user data. With the master key, your Cloud Code function can override any ACLs and write data. This means that it'll bypass all the security mechanisms you've put in place in the previous sections.

Say you want to allow a user to “like” a `Post` object without giving them full write permissions on the object. You can do this by having the client call a Cloud Code function instead of modifying the Post itself:

The master key should be used carefully. When invoked, the master key is in effect for the duration of the Cloud Code function in which it is called:

```
Parse.Cloud.define("like", function(request, response) {
  Parse.Cloud.useMasterKey();
  // Everything after this point will bypass ACLs
  // and other security
  // even if I do things besides just updating a
  // Post object.
});
```

A more prudent way to use the master key would be to pass it as a parameter on a per-function basis. For example, instead of the above, set `useMasterKey` to `true` in each individual API function:

```
Parse.Cloud.define("like", function(request, response) {
  var post = new Parse.Object("Post");
  post.id = request.params.postId;
  post.increment("likes");
  post.save(null, { useMasterKey: true
}).then(function() {
  // If I choose to do something else here, it
  // won't be using
  // the master key and I'll be subject to
  // ordinary security measures.
  response.success();
}, function(error) {
  response.error(error);
});
});
```

One very common use case for Cloud Code is sending push notifications to particular users. In general, clients can't be trusted to send push notifications directly, because they could modify the alert text, or push to people they shouldn't be able to. Your app's settings will allow you to set whether "client push" is enabled or not; we recommend that you make sure it's disabled. Instead, you should write Cloud Code functions that validate the data to be pushed and sent before sending a push.

Parse Security Summary

Parse provides a number of ways for you to secure data in your app. As you build your app and evaluate the

kinds of data you will be storing, you can make the decision about which implementation to choose.

It is worth repeating that the Parse User object is readable by all other users by default. You will want to set the ACL on your User object accordingly if you wish to prevent data contained in the User object (for example, the user's email address) from being visible by other users.

Most classes in your app will fall into one of a couple of easy-to-secure categories. For fully public data, you can use class-level permissions to lock down the table to put publicly readable and writeable by no one. For fully private data, you can use ACLs to make sure that only the user who owns the data can read it. But occasionally, you'll run into situations where you don't want data that's fully public or fully private. For example, you may have a social app, where you have data for a user that should be readable only to friends whom they've approved. For this you'll need to a combination of the techniques discussed in this guide to enable exactly the sharing rules you desire.

We hope that you'll use these tools to do everything you can to keep your app's data and your users' data secure. Together, we can make the web a safer place.

Want to contribute to this doc? [Edit this section.](#)

Performance

As your app scales, you will want to ensure that it performs well under increased load and usage. This document provides guidelines on how you can optimize your app's performance. While you can use Parse Server for quick prototyping and not worry about performance, you will want to keep our performance guidelines in mind when you're initially designing your app. We strongly advise that you make sure you've followed all suggestions before releasing your app.

You can improve your app's performance by looking at the following:

- + Writing efficient queries.

- + Writing restrictive queries.
- + Using client-side caching.
- + Using Cloud Code.
- + Avoiding count queries.
- + Using efficient search techniques.

Keep in mind that not all suggestions may apply to your app. Let's look into each one of these in more detail.

Write Efficient Queries

Parse objects are stored in a database. A Parse query retrieves objects that you are interested in based on conditions you apply to the query. To avoid looking through all the data present in a particular Parse class for every query, the database can use an index. An index is a sorted list of items matching a given criteria. Indexes help because they allow the database to do an efficient search and return matching results without looking at all of the data. Indexes are typically smaller in size and available in memory, resulting in faster lookups.

Indexing

You are responsible for managing your database and maintaining indexes when using Parse Server. If your data is not indexed, every query will have to go through the the entire data for a class to return a query result. On the other hand, if your data is indexed appropriately, the number of documents scanned to return a correct query result should be low.

The order of a query constraint's usefulness is:

- + Equal to
- + Contained In
- + Less than, Less than or Equal to, Greater than, Greater than or Equal to
- + Prefix string matches
- + Not equal to
- + Not contained in
- + Everything else

Take a look at the following query to retrieve GameScore objects:

```
var names = new[] { "Jonathan Walsh", "Dario Wunsch", "Shawn Simon" };  
var query = new ParseObject.GetQuery("GameScore")  
    .WhereEqualTo("score", 50)  
    .WhereContainedIn("playerName", names);
```

Creating an index query based on the score field would yield a smaller search space in general than creating one on the `playerName` field.

When examining data types, booleans have a very low entropy and do not make good indexes. Take the following query constraint:

```
query.WhereEqualTo("cheatMode", false);
```

The two possible values for `"cheatMode"` are `true` and `false`. If an index was added on this field it would be of little use because it's likely that 50% of the records will have to be looked at to return query results.

Data types are ranked by their expected entropy of the value space for the key:

- + GeoPoints
- + Array
- + Pointer
- + Date
- + String
- + Number
- + Other

Even the best indexing strategy can be defeated by suboptimal queries.

Efficient Query Design

Writing efficient queries means taking full advantage of indexes. Let's take a look at some query constraints that negate the use of indexes:

- + Not Equal To
- + Not Contained In

Additionally, the following queries under certain scenarios may result in slow query responses if they can't take advantage of indexes:

- + Regular Expressions
- + Ordered By

NOT EQUAL TO

For example, let's say you're tracking high scores for a game in a `GameScore` class. Now say you want to retrieve the scores for all players except a certain one. You could create this query:

```
var results = await
ParseObject.GetQuery("GameScore")
    .WhereNotEqualTo("playerName", "Michael
Yabuti")
    .FindAsync();
```

This query can't take advantage of indexes. The database has to look at all the objects in the `"GameScore"` class to satisfy the constraint and retrieve the results. As the number of entries in the class grows, the query takes longer to run.

Luckily, most of the time a "Not Equal To" query condition can be rewritten as a "Contained In" condition. Instead of querying for the absence of values, you ask for values which match the rest of the column values. Doing this allows the database to use an index and your queries will be faster.

For example if the `User` class has a column called `state` which has values "SignedUp", "Verified", and "Invited", the slow way to find all users who have used the app at least once would be to run the query:

```
var query = ParseUser.Query
    .WhereNotEqualTo("state", "Invited");
```

It would be faster to use the "Contained In" condition when setting up the query:

```
query.WhereContainedIn("state", new[] {  
    "SignedUp", "Verified" });
```

Sometimes, you may have to completely rewrite your query. Going back to the `"GameScore"` example, let's say we were running that query to display players who had scored higher than the given player. We could do this differently, by first getting the given player's high score and then using the following query:

```
// Previously retrieved highScore for Michael  
Yabuti  
var results = await  
ParseObject.GetQuery("GameScore")  
    .WhereGreaterThan("score", highScore)  
    .FindAsync();
```

The new query you use depends on your use case. This may sometimes mean a redesign of your data model.

NOT CONTAINED IN

Similar to “Not Equal To”, the “Not Contained In” query constraint can't use an index. You should try and use the complementary “Contained In” constraint. Building on the User example, if the state column had one more value, “Blocked”, to represent blocked users, a slow query to find active users would be:

```
var query = ParseUser.Query  
    .WhereNotContainedIn("state", new[] {  
        "Invited", "Blocked" });
```

Using a complimentary “Contained In” query constraint will always be faster:

```
query.WhereContainedIn("state", new[] {  
    "SignedUp", "Verified"});
```

This means rewriting your queries accordingly. Your query rewrites will depend on your schema set up. It may mean redoing that schema.

REGULAR EXPRESSIONS

Regular expression queries should be avoided due to performance considerations. MongoDB is not efficient for doing partial string matching except for the special

case where you only want a prefix match. Queries that have regular expression constraints are therefore very expensive, especially for classes with over 100,000 records. Consider restricting how many such operations can be run on a particular app at any given time.

You should avoid using regular expression constraints that don't use indexes. For example, the following query looks for data with a given string in the `"playerName"` field. The string search is case insensitive and therefore cannot be indexed:

```
query.WhereMatches("playerName", "Michael", "i")
```

The following query, while case sensitive, looks for any occurrence of the string in the field and cannot be indexed:

```
query.WhereContains("playerName", "Michael")
```

These queries are both slow. In fact, the `matches` and `contains` query constraints are not covered in our querying guides on purpose and we do not recommend using them. Depending on your use case, you should switch to using the following constraint that uses an index, such as:

```
query.WhereStartsWith("playerName", "Michael")
```

This looks for data that starts with the given string. This query will use the backend index, so it will be faster even for large datasets.

As a best practice, when you use regular expression constraints, you'll want to ensure that other constraints in the query reduce the result set to the order of hundreds of objects to make the query efficient. If you must use the `matches` or `contains` constraints for legacy reasons, then use case sensitive, anchored queries where possible, for example:

```
query.WhereMatches("playerName", "^Michael")
```

Most of the use cases around using regular expressions involve implementing search. A more performant way of implementing search is detailed later.

Write Restrictive Queries

Writing restrictive queries allows you to return only the data that the client needs. This is critical in a mobile environment where data usage can be limited and network connectivity unreliable. You also want your mobile app to appear responsive and this is directly affected by the objects you send back to the client. The [Querying section](#) shows the types of constraints you can add to your existing queries to limit the data returned. When adding constraints, you want to pay attention and design efficient queries.

You can limit the number of query results returned. The limit is 100 by default but anything from 1 to 1000 is a valid limit:

```
query.Limit(10); // limit to at most 10 results
```

If you're issuing queries on GeoPoints, make sure you specify a reasonable radius:

```
var results = await
ParseObject.GetQuery("GameScore")
    .WhereWithinDistance("location",
userGeoPoint, ParseGeoDistance.FromMiles(10.0))
    .FindAsync();
```

You can further limit the fields returned by calling select:

```
var results = await
ParseObject.GetQuery("GameScore")
    .Select(new[] { "score", "playerName" })
    .FindAsync();
// each of results will only have the selected
fields available.
```

Client-side Caching

For queries run from iOS and Android, you can turn on query caching. See the [iOS](#) and [Android](#) guides for more details. Caching queries will increase your mobile app's performance especially in cases where you want to display cached data while fetching the latest data from Parse.

Use Cloud Code

Cloud Code allows you to run custom JavaScript logic on Parse Server instead of on the client.

You can use this to offload processing to the Parse servers thus increasing your app's perceived performance. You can create hooks that run whenever an object is saved or deleted. This is useful if you want to validate or sanitize your data. You can also use Cloud Code to modify related objects or kick off other processes such as sending off a push notification.

We saw examples of limiting the data returned by writing restrictive queries. You can also use [Cloud Functions](#) to help limit the amount of data returned to your app. In the following example, we use a Cloud Function to get a movie's average rating:

```
Parse.Cloud.define("averageStars",
function(request, response) {
  var Review = Parse.Object.extend("Review");
  var query = new Parse.Query(Review);
  query.equalTo("movie", request.params.movie);
  query.find().then(function(results) {
    var sum = 0;
    for (var i = 0; i < results.length; ++i) {
      sum += results[i].get("stars");
    }
    response.success(sum / results.length);
  }, function(error) {
    response.error("movie lookup failed");
  });
});
```

You could have ran a query on the Review class on the client, returned only the stars field data and computed the result on the client. As the number of reviews for a movie increases you can see that the data being returned to the device using this methodology also increases. Implementing the functionality through a Cloud Function returns the one result if successful.

As you look at optimizing your queries, you'll find that you may have to change the queries - sometimes even after you've shipped your app to the App Store or Google Play. The ability to change your queries without a client update is possible if you use [Cloud Functions](#). Even if you have to redesign your schema, you could

make all the changes in your Cloud Functions while keeping the client interface the same to avoid an app update. Take the average stars Cloud Function example from before, calling it from a client SDK would look like this:

```
IDictionary<string, object> dictionary = new
Dictionary<string, object>
{
    { "movie", "The Matrix" }
};

ParseCloud.CallFunctionAsync<float>
("averageStars", dictionary).ContinueWith(t => {
    var result = t.Result;
    // result is 4.5
});
```

If later on, you need to modify the underlying data model, your client call can remain the same, as long as you return back a number that represents the ratings result.

Avoid Count Operations

For classes with over 1,000 objects, count operations are limited by timeouts. Thus, it is preferable to architect your application to avoid this count operation.

Suppose you are displaying movie information in your app and your data model consists of a `Movie` class and a `Review` class that contains a pointer to the corresponding movie. You might want to display the review count for each movie on the top-level navigation screen using a query like this:

```
var count = await ParseObject.GetQuery("Review")
// movieId corresponds to a given movie's id
    .WhereEqualTo("movie", movieId)
    .CountAsync();
```

If you run the count query for each of the UI elements, they will not run efficiently on large data sets. One approach to avoid using the `count()` operator could be to add a field to the `Movie` class that represents the review count for that movie. When saving an entry to the `Review` class you could increment the

corresponding movie's review count field. This can be done in an `afterSave` handler:

```
Parse.Cloud.afterSave("Review",
function(request) {
  // Get the movie id for the Review
  var movieId = request.object.get("movie").id;
  // Query the Movie represented by this review
  var Movie = Parse.Object.extend("Movie");
  var query = new Parse.Query(Movie);
  query.get(movieId).then(function(movie) {
    // Increment the reviews field on the Movie
    object
    movie.increment("reviews");
    movie.save();
  }, function(error) {
    throw "Got an error " + error.code + " : " +
error.message;
  });
});
```

Your new optimized query would not need to look at the Review class to get the review count:

```
var results = await ParseObject.GetQuery("Movie")
  .FindAsync();
// Results include the reviews count field
```

You could also use a separate Parse Object to keep track of counts for each review. Whenever a review gets added or deleted, you can increment or decrement the counts in an `afterSave` or `afterDelete` Cloud Code handler. The approach you choose depends on your use case.

Implement Efficient Searches

As mentioned previously, MongoDB is not efficient for doing partial string matching. However, this is an important use case when implementing search functionality that scales well in production.

Simplistic search algorithms simply scan through all the class data and executes the query on each entry. The key to making searches run efficiently is to minimize the number of data that has to be examined when executing each query by using an index as we've outlined earlier. You'll need to build your data model in a way that it's easy for us to build an index for the data

you want to be searchable. For example, string matching queries that don't match an exact prefix of the string won't be able to use an index leading to timeout errors as the data set grows.

Let's walk through an example of how you could build an efficient search. You can apply the concepts you learn in this example to your use case. Say your app has users making posts, and you want to be able to search those posts for hashtags or particular keywords. You'll want to pre-process your posts and save the list of hashtags and words into array fields. You can do this processing either in your app before saving the posts, or you can use a Cloud Code `beforeSave` hook to do this on the fly:

```
var _ = require("underscore");
Parse.Cloud.beforeSave("Post", function(request,
response) {
  var post = request.object;
  var toLowerCase = function(w) { return
w.toLowerCase(); };
  var words = post.get("text").split(/\b/);
  words = _.map(words, toLowerCase);
  var stopWords = ["the", "in", "and"]
  words = _.filter(words, function(w) {
    return w.match(/^\w+$/) && !
_.contains(stopWords, w);
  });
  var hashtags = post.get("text").match(/#.+?
\b/g);
  hashtags = _.map(hashtags, toLowerCase);
  post.set("words", words);
  post.set("hashtags", hashtags);
  response.success();
});
```

This saves your words and hashtags in array fields, which MongoDB will store with a multi-key index. There are some important things to notice about this. First of all it's converting all words to lower case so that we can look them up with lower case queries, and get case insensitive matching. Secondly, it's filtering out common words like 'the', 'in', and 'and' which will occur in a lot of posts, to additionally reduce useless scanning of the index when executing the queries.

Once you've got the keywords set up, you can efficiently look them up using "All" constraint on your

query:

```
var Post = Parse.Object.extend("Post");
var query = new Parse.Query(Post);
query.containsAll("hashtags", ["#parse",
"#ftw"]);
query.find().then(function(results) {
    // Request succeeded
}, function(error) {
    // Request failed
});
```

```
var results = await ParseObject.GetQuery("Post")
    .WhereContainsAll("hashtags", new[] {
"#parse", "#ftw" })
    .FindAsync();
```

Limits and Other Considerations

There are some limits in place to ensure the API can provide the data you need in a performant manner. We may adjust these in the future. Please take a moment to read through the following list:

Objects

- + Parse Objects are limited in size to 128 KB.
- + We recommend against creating more than 64 fields on a single Parse Object to ensure that we can build effective indexes for your queries.
- + We recommend against using field names that are longer than 1,024 characters, otherwise an index for the field will not be created.

Queries

- + Queries return 100 objects by default. Use the `limit` parameter to change this, up to a value of 1,000.
- + Queries can only return up to 1,000 objects in a single result set. This includes any resolved pointers. You can use `skip` and `limit` to page through results.
- + The maximum value accepted by `skip` is 10,000. If you need to get more objects, we recommend sorting the results and then using a constraint on the sort column to filter out the first 10,000 results. You will then be able to continue paging through results starting from a `skip` value of 0.

For example, you can sort your results by `createdAt ASC` and then filter out any objects older than the `createdAt` value of the 10,000th object when starting again from 0.

- + Alternatively, you may use the `each()` method in the JavaScript SDK to page through all objects that match the query.
- + Skips and limits can only be used on the outer query.
- + You may increase the limit of a inner query to 1,000, but skip cannot be used to get more results past the first 1,000.
- + Constraints that collide with each other will result in only one of the constraint being applied. An example of this would be two `equalTo` constraints over the same key with two different values, which contradicts itself (perhaps you're looking for 'contains').
- + No geo-queries inside compound OR queries.
- + Using `$exists: false` is not advised.
- + The `each` query method in the JavaScript SDK cannot be used in conjunction with queries using geo-point constraints.
- + A maximum of 500,000 objects will be scanned per query. If your constraints do not successfully limit the scope of the search, this can result in queries with incomplete results.
- + A `containsAll` query constraint can only take up to 9 items in the comparison array.

Push Notifications

- + **Delivery of notifications is a “best effort”, not guaranteed.** It is not intended to deliver data to your app, only to notify the user that there is new data available.

Cloud Code

- + The `params` payload that is passed to a Cloud Function is limited to 50 MB.

Want to contribute to this doc? [Edit this section.](#)

Error Codes

The following is a list of all the error codes that can be returned by the Parse API. You may also refer to [RFC2616](#) for a list of http error codes. Make sure to check the error message for more details.

API Issues

| Name | Code | Description |
|------------------------|------|---|
| UserInvalidLoginParams | 101 | Invalid login parameters. Check error message for more details. |
| ObjectNotFound | 101 | The specified object or session doesn't exist or could not be found. Can also indicate that you do not have the necessary permissions to read or write this object. Check error message for more details. |
| InvalidQuery | 102 | There is a problem with the parameters used to construct this query. This could be an invalid field name or an invalid field type for a specific constraint. Check error message for more details. |
| InvalidClassName | 103 | Missing or invalid classname. Classnames are case-sensitive. They must start with a letter, and a-zA-Z0-9_ are the only valid characters. |
| MissingObjectId | 104 | An unspecified object id. |

| Name | Code | Description |
|---------------------------------|------|---|
| <code>InvalidFieldName</code> | 105 | An invalid field name. Keys are case-sensitive. They must start with a letter, and a-zA-Z0-9_ are the only valid characters. Some field names may be reserved. Check error message for more details. |
| <code>InvalidPointer</code> | 106 | A malformed pointer was used. You would typically only see this if you have modified a client SDK. |
| <code>InvalidJSON</code> | 107 | Badly formed JSON was received upstream. This either indicates you have done something unusual with modifying how things encode to JSON, or the network is failing badly. Can also indicate an invalid utf-8 string or use of multiple form encoded values. Check error message for more details. |
| <code>CommandUnavailable</code> | 108 | The feature you tried to access is only available internally for testing purposes. |
| <code>NotInitialized</code> | 109 | You must call <code>Parse.initialize</code> before using the Parse library. Check the Quick Start guide for your platform. |
| <code>ObjectTooLarge</code> | 116 | The object is too large. Parse Objects have a max size of 128 kilobytes. |

| Name | Code | Description |
|---------------------------|------|---|
| ExceededConfigParamsError | 116 | You have reached the limit of 100 config parameters. |
| InvalidLimitError | 117 | An invalid value was set for the limit. Check error message for more details. |
| InvalidSkipError | 118 | An invalid value was set for skip. Check error message for more details. |
| OperationForbidden | 119 | The operation isn't allowed for clients due to class-level permissions. Check error message for more details. |
| CacheMiss | 120 | The result was not found in the cache. |
| InvalidNestedKey | 121 | An invalid key was used in a nested JSONObject. Check error message for more details. |
| InvalidACL | 123 | An invalid ACL was provided. |
| InvalidEmailAddress | 125 | The email address was invalid. |
| DuplicateValue | 137 | Unique field was given a value that is already taken. |
| InvalidRoleName | 139 | Role's name is invalid. |
| ReservedValue | 139 | Field value is reserved. |
| ExceededCollectionQuota | 140 | You have reached the quota on the number of classes in your app. Please delete some classes if you need to add a new class. |

| Name | Code | Description |
|------------------------------------|------|---|
| <code>ScriptFailed</code> | 141 | Cloud Code script failed. Usually points to a JavaScript error. Check error message for more details. |
| <code>FunctionNotFound</code> | 141 | Cloud function not found. Check that the specified Cloud function is present in your Cloud Code script and has been deployed. |
| <code>JobNotFound</code> | 141 | Background job not found. Check that the specified job is present in your Cloud Code script and has been deployed. |
| <code>SuccessErrorNotCalled</code> | 141 | success/error was not called. A cloud function will return once <code>response.success()</code> or <code>response.error()</code> is called. A background job will similarly finish execution once <code>status.success()</code> or <code>status.error()</code> is called. If a function or job never reaches either of the success/error methods, this error will be returned. This may happen when a function does not handle an error response correctly, preventing code execution from reaching the <code>success()</code> method call. |

| Name | Code | Description |
|--|------|--|
| <code>MultiplesuccessErrorCalls</code> | 141 | Can't call success/error multiple times. A cloud function will return once response.success() or response.error() is called. A background job will similarly finish execution once status.success() or status.error() is called. If a function or job calls success() and/or error() more than once in a single execution path, this error will be returned. |
| <code>ValidationFailed</code> | 142 | Cloud Code validation failed. |
| <code>WebhookError</code> | 143 | Webhook error. |
| <code>InvalidImageData</code> | 150 | Invalid image data. |
| <code>UnsavedFileError</code> | 151 | An unsaved file. |
| <code>InvalidPushTimeError</code> | 152 | An invalid push time was specified. |
| <code>HostingError</code> | 158 | Hosting error. |
| <code>InvalidEventName</code> | 160 | The provided analytics event name is invalid. |
| <code>ClassNotEmpty</code> | 255 | Class is not empty and cannot be dropped. |
| <code>AppNameInvalid</code> | 256 | App name is invalid. |
| <code>MissingAPIKeyError</code> | 902 | The request is missing an API key. |
| <code>InvalidAPIKeyError</code> | 903 | The request is using an invalid API key. |

Push related errors

| Name | Code | Description |
|--------------------------------------|------|---|
| <code>IncorrectType</code> | 111 | A field was set to an inconsistent type. Check error message for more details. |
| <code>InvalidChannelName</code> | 112 | Invalid channel name. A channel name is either an empty string (the broadcast channel) or contains only a-zA-Z0-9_ characters and starts with a letter. |
| <code>InvalidSubscriptionType</code> | 113 | Bad subscription type. Check error message for more details. |
| <code>InvalidDeviceToken</code> | 114 | The provided device token is invalid. |
| <code>PushMisconfigured</code> | 115 | Push is misconfigured in your app. Check error message for more details. |
| <code>PushWhereAndChannels</code> | 115 | Can't set channels for a query-targeted push. You can fix this by moving the channels into your push query constraints. |

| Name | Code | Description |
|---|------|--|
| <code>PushWhereAndType</code> | 115 | Can't set device type for a query-targeted push. You can fix this by incorporating the device type constraints into your push query. |
| <code>PushMissingData</code> | 115 | Push is missing a 'data' field. |
| <code>PushMissingChannels</code> | 115 | Non-query push is missing a 'channels' field. Fix by passing a 'channels' or 'query' field. |
| <code>ClientPushDisabled</code> | 115 | Client-initiated push is not enabled. Check your Parse app's push notification settings. |
| <code>RestPushDisabled</code> | 115 | REST-initiated push is not enabled. Check your Parse app's push notification settings. |
| <code>ClientPushWithURI</code> | 115 | Client-initiated push cannot use the "uri" option. |
| <code>PushQueryOrPayloadTooLarge</code> | 115 | Your push query or data payload is too large. Check error message for more details. |

| Name | Code | Description |
|-------------------------------------|------|---|
| <code>InvalidExpirationError</code> | 138 | Invalid expiration value. |
| <code>MissingPushIdError</code> | 156 | A push id is missing. Deprecated. |
| <code>MissingDeviceTypeError</code> | 157 | The device type field is missing. Deprecated. |

File related errors

| Name | Code | Description |
|-----------------------------------|------|---|
| <code>InvalidFileName</code> | 122 | An invalid filename was used for Parse File. A valid file name contains only a-zA-ZO-9_ characters and is between 1 and 128 characters. |
| <code>MissingContentType</code> | 126 | Missing content type. |
| <code>MissingContentLength</code> | 127 | Missing content length. |
| <code>InvalidContentLength</code> | 128 | Invalid content length. |
| <code>FileTooLarge</code> | 129 | File size exceeds maximum allowed. |
| <code>FileSaveError</code> | 130 | Error saving a file. |
| <code>FileDeleteError</code> | 131 | File could not be deleted. |

Installation related errors

| Name | Code | Description |
|------|------|-------------|
|------|------|-------------|

| | | |
|----------------------------|-----|---------------------------------|
| InvalidInstallationIdError | 132 | Invalid installation id. |
| InvalidDeviceTypeError | 133 | Invalid device type. |
| InvalidChannelsArrayError | 134 | Invalid channels array value. |
| MissingRequiredFieldError | 135 | Required field is missing. |
| ChangedImmutableFieldError | 136 | An immutable field was changed. |

Purchase related errors

| Name | Code | Description |
|---------------------------|------|--|
| ReceiptMissing | 143 | Product purchase receipt is missing. |
| InvalidPurchaseReceipt | 144 | Product purchase receipt is invalid. |
| PaymentDisabled | 145 | Payment is disabled on this device. |
| InvalidProductIdentifier | 146 | The product identifier is invalid. |
| ProductNotFoundInAppStore | 147 | The product is not found in the App Store. |
| InvalidServerResponse | 148 | The Apple server response is not valid. |

| Name | Code | Description |
|--------------------------------|------|---|
| ProductDownloadFilesystemError | 149 | The product fails to download due to file system error. |

User related errors

| Name | Code | Description |
|-----------------------------|------|---|
| UsernameMissing | 200 | The username is missing or empty. |
| PasswordMissing | 201 | The password is missing or empty. |
| UsernameTaken | 202 | The username has already been taken. |
| UserEmailTaken | 203 | Email has already been used. |
| UserEmailMissing | 204 | The email is missing, and must be specified. |
| UserWithEmailNotFound | 205 | A user with the specified email was not found. |
| SessionMissing | 206 | A user object without a valid session could not be altered. |
| MustCreateUserThroughSignup | 207 | A user can only be created through signup. |

| Name | Code | Description |
|----------------------|------|---|
| AccountAlreadyLinked | 208 | An account being linked is already linked to another user. |
| InvalidSessionToken | 209 | The device's session token is no longer valid. The application should ask the user to log in again. |

Linked services errors

| Name | Code | Description |
|------------------------|------|---|
| LinkedIdMissing | 250 | A user cannot be linked to an account because that account's id could not be found. |
| InvalidLinkedSession | 251 | A user with a linked (e.g. Facebook or Twitter) account has an invalid session. Check error message for more details. |
| InvalidGeneralAuthData | 251 | Invalid auth data value used. |
| BadAnonymousID | 251 | Anonymous id is not a valid lowercase UUID. |
| FacebookBadToken | 251 | The supplied Facebook session token is expired or invalid. |

| Name | Code | Description |
|---------------------------|------|--|
| FacebookBadID | 251 | A user with a linked Facebook account has an invalid session. |
| FacebookWrongAppID | 251 | Unacceptable Facebook application id. |
| TwitterVerificationFailed | 251 | Twitter credential verification failed. |
| TwitterWrongID | 251 | Submitted Twitter id does not match the id associated with the submitted access token. |
| TwitterWrongScreenName | 251 | Submitted Twitter handle does not match the handle associated with the submitted access token. |
| TwitterConnectFailure | 251 | Twitter credentials could not be verified due to problems accessing the Twitter API. |
| UnsupportedService | 252 | A service being linked (e.g. Facebook or Twitter) is unsupported. Check error message for more details. |
| UsernameSigninDisabled | 252 | Authentication by username and password is not supported for this application. Check your Parse app's authentication settings. |

| Name | Code | Description |
|---------------------------------------|------|---|
| <code>AnonymousSigninDisabled</code> | 252 | Anonymous users are not supported for this application. Check your Parse app's authentication settings. |
| <code>FacebookSigninDisabled</code> | 252 | Authentication by Facebook is not supported for this application. Check your Parse app's authentication settings. |
| <code>TwitterSigninDisabled</code> | 252 | Authentication by Twitter is not supported for this application. Check your Parse app's authentication settings. |
| <code>InvalidAuthDataError</code> | 253 | An invalid authData value was passed. Check error message for more details. |
| <code>LinkingNotSupportedError</code> | 999 | Linking to an external account not supported yet with <code>signup_or_login</code> . Use <code>update</code> instead. |

Client-only errors

| Name | Code | Description |
|-------------------------------|------|---|
| <code>ConnectionFailed</code> | 100 | The connection to the Parse servers failed. |

| Name | Code | Description |
|-----------------------------|------|--|
| <code>AggregateError</code> | 600 | There were multiple errors. Aggregate errors have an “errors” property, which is an array of error objects with more detail about each error that occurred. |
| <code>FileReadError</code> | 601 | Unable to read input for a Parse File on the client. |
| <code>XDomainRequest</code> | 602 | A real error code is unavailable because we had to use an <code>XDomainRequest</code> object to allow CORS requests in Internet Explorer, which strips the body from HTTP responses that have a non-2XX status code. |

Operational issues

| Name | Code | Description |
|---------------------------------------|------|--|
| <code>RequestTimeout</code> | 124 | The request was slow and timed out. Typically this indicates that the request is too expensive to run. You may see this when a Cloud function did not finish before timing out, or when a <code>Parse.Cloud.httpRequest</code> connection times out. |
| <code>InefficientQueryError</code> | 154 | An inefficient query was rejected by the server. Refer to the Performance Guide and slow query log. |
| <code>RequestLimitExceeded</code> | 155 | This application has exceeded its request limit (legacy Parse.com apps only). |
| <code>TemporaryRejectionError</code> | 159 | An application’s requests are temporary rejected by the server (legacy Parse.com apps only). |
| <code>DatabaseNotMigratedError</code> | 428 | You should migrate your database as soon as possible (legacy Parse.com apps only). |

Other issues

| Name | Code | Description |
|---------------------|------|---|
| OtherCause | -1 | An unknown error or an error unrelated to Parse occurred. |
| InternalServerError | 1 | Internal server error. No information available. |
| ServiceUnavailable | 2 | The service is currently unavailable. |
| ClientDisconnected | 4 | Connection failure. |

Want to contribute to this doc? [Edit this section.](#)
