Farse — A Parse Clone (Fake Parse)

EXECUTIVE SUMMARY

Objective

To model a proof of concept for a Parse-type service clone, in order to serve as a rapid prototyping backend for mobile apps.

Goals

The goals of the project are to have a schema-less high-performance database for use with mobile applications, characterized by minimal configuration and flexible operation.

Solution

The proposed solution combines any number of thin clients, written in any language, capable of posting JSON data via HTTP to a REST api-style backend. The included demo / proof of concept presents a thin client written in Objective C which models the Parse Objective C / iOS object model. The server backend is modeled using the Apache Solr project running in the Jetty servlet container. As such, this demo can be run on any current macintosh with Xcode 6 and a current java runtime environment.

Apache Solr Backend

The Apache Software Foundation's Solr (pronounced "solar" engine is a high performance document-oriented database which features many of the requested features of the system. Among these features are:

- Schema-less operation
- Rest API over HTTP
- Paging result sets
- Highly performant concurrency implementation
- Runs in java servlet container - the engine installs as a jar file
- Web administration interface / dashboard with query tools

Additional features of the solr backend are:

- Sharded caching
- Strict schema operation
- Automatic indexing, nearly real-time
- Linearly scalable - auto index replication,
- Auto failover and recovery
- Flexible configuration through xml files
- Extensible plugin architecture
- Multiple indexes
- Geospatial search
- Configurable, user-level caching
- Autowarming
- Transaction logging
- Automated failover
- Integration with Apache Camel

Client APIs

The flexible nature of the Solr backend is such that any language which is capable of REST queries can interface with the backend, using the Solr Query Langage, a rich language which supports field-level primitive, range, and faceted queries, among others. Separate endpoints are provided for saving and querying data using JSON or XML data over HTTP. A client emulates the rest query api of Solr, for both saving and querying, and wraps it in an api using their native language

Supported Datatypes

Field types in Solr are configurable and can be extended to support many complex datatypes. Datatypes provided by the default solr engine are many, in fact too many to list here. Common primitive types supported by default are:

- Integer
- Short Integer
- Float
- Long
- Double
- String
- Text
- Date
- Enum
- Binary
- Boolean
- Byte

In addition, many useful complex datatypes are supported by default as well, such as:

- Latitude/Longitude
- Currency
- UDID
- Trie
- Spatial Recursive Prefix Tree

Note: In schemaless operation, primitives types are inferred for standard data types. This can be configured to accommodate more complex implementations, and it is recommended that in a production environment, specific schemas would be created and configured to handle the specific needs of an application.

Example Objective C Client

The provided thin client implementation of this demo is written for iOS / Cocoa in Objective C language. The example Farse client closely models the published Parse iOS api. Included are the following objects and methods:

Creating Objects

FPObject - models PFObject

Provided object creation methods:

```
/**
 * create an object with the specified classname
 * @param[in] className - an NSString instance naming the class
 */
+(FPObject*) objectWithClassName:(NSString*) className;


/**
 * Create an FPobject using the specified dictionary as the backing store.
 * This is the method used to convert raw json data into FPObjects
 */
+(FPObject*) objectWithDictionary:(NSDictionary*) dictionary;


/**
 * Asynchronously save the object to the solr backend
 */
-(void) saveInBackground;
```

Querying Objects

FPQuery - models PFQuery

Provided object query methods:

```
/**
 * create a query object targeted towards the specified class
 * @param[in] className - an NSString instance describing the class to query
 */
+(FPQuery*) queryWithClassName:(NSString*) className;

/**
 * Retrieve a single object by specifying its primary key,
 * (aka "id" field in Solr, "objectId", in FPObject.
 * Solr, in schema-less mode, will not need the className parameter
 * to return an object with this method.
 * In Parse, this method's signature returns the found object.
```

```
 * In FP the object is queried asynchonously,
 * so it uses a completion block to return the object instead.
 */
-(void) getObjectInBackgroundWithId:(NSString*) objectId
         block:(void(^)(FPObject *object, NSError *error) ) completionBlock;

/**
 * set criteria for this query's query request
 * @param[in] whereKey: the name of the field to search in for "match"
 * @param[in] equalTo: the value of the field specified in whereKey to match
 */
-(void) whereKey:(NSString*)whereClause equalTo:(NSString*)match;

/**
 * Execute a query, returning an array of FPObjects
 *
 */
-(void)findObjectsInBackgroundWithBlock:(void(^)(NSArray *objects, NSError
*error) ) completionBlock;
```

Example source using the object creation methods:

```
    // create the object, giving it a "class name".
    // Using Parse, the object would be stored in a "table" with the name of the class.
    // In Solr, the object class name becomes a single field in each "document", named "className".
    // To facilitate retrieval within a single "table', queries can make use of Solr's faceted search
    // capabilities to cordon off a series of documents matching a specific "object type".

    FPObject *object = [FPObject objectWithClassName:@"GameScore"];

    // set values on the object, using string keys, which become field names in Solr
    // values can be typical primative json types, such as string, numbers, and boolean.
    object[@"score"] = @1337;
    object[@"cheatMode"] = @NO;
    object[@"playerName"] = @"Sean Plott";

    // save the object asynchronously to the solr backend
    [object saveInBackground];
```

In the preceding example, the object with name “Sean Plott” is created on the fly, with arbitrary fields added. It is then saved using an asynchronous network call, which posts the data to solr as a document, and is indexed immediately making it ready for querying as in the next example.

Example source using the object query methods:

In this example, we retrieve the previously made example in three ways: as an individual object, specifying the object's id; secondly, by finding multiple objects of the same "className", which is effectively an object type, or can be thought of as a "table". Thirdly, the objects are retrieved using an additional "where clause", to refine the query against object type "GameScore". The third example makes use of Solr's faceted querying.

```
     // find a single object using the specified object id
FPQuery *query = [FPQuery queryWithClassName:@"GameScore"];
[query getObjectInBackgroundWithId:self.lastObjectId block:^(FPObject *object, NSError *error) {
    NSLog(@"object by id query complete %@",object);
    NSLog(@"name %@ score %@",object[@"playerName"],object[@"score"]);
}];

//alternatively  find all objects with field name GameScore
[query findObjectsInBackgroundWithBlock:^(NSArray *objects, NSError *error) {
    NSLog(@"multiple objects query complete: found %d object(s)",objects.count);
    for (FPObject *o in objects) {
        NSLog(@"object: %@ - %@",o[@"playerName"],o[@"score"]);
    }
}];

// now do the same query but add a "where clause"
[query whereKey:@"playerName" equalTo:@"Sean Plott"];

[query findObjectsInBackgroundWithBlock:^(NSArray *objects, NSError *error) {
    NSLog(@"multiple objects query complete: found %d object(s)",objects.count);
    for (FPObject *o in objects) {
        NSLog(@"object: %@ - %@",o[@"playerName"],o[@"score"]);
    }
}];
```

Object IDs

Objects require unique ids in solr. In schema-less mode, Solr will then require a unique id across the entire collection, which means that a UUID-style string (BBCAE0D4-9133-418F-9BE4-D4A8814A6DC3) is recommended in order to avoid possible data collisions. In the demo code, the object uses NSObject's built-in 'hash' value.

Creating and saving multiple objects with the same data in an app will result in duplicate entries, although with differing primary keys. This is because the id generated per object is based on a hash of the object itself.
To update data in an existing FPObject, it is required that you retrieve the object from Solr and use its id in future 'save' operations. This behavior mirrors Parse, as noted here: https://www.parse.com/questions/multiple-object-ids

INSTALLATION INSTRUCTIONS

Prerequisites:

- A working java runtime installation. This demo was tested against JDK 1.7.0_65.
- Xcode 6

Steps:

1) Download the solr engine from https://lucene.apache.org/solr/downloads.html and unzip to some location.

2) Locate the "example" subdirectory directory, and in Terminal, cd to it, i.e.:

```
cd ~/Downloads/solr-4.6.1/example/
```

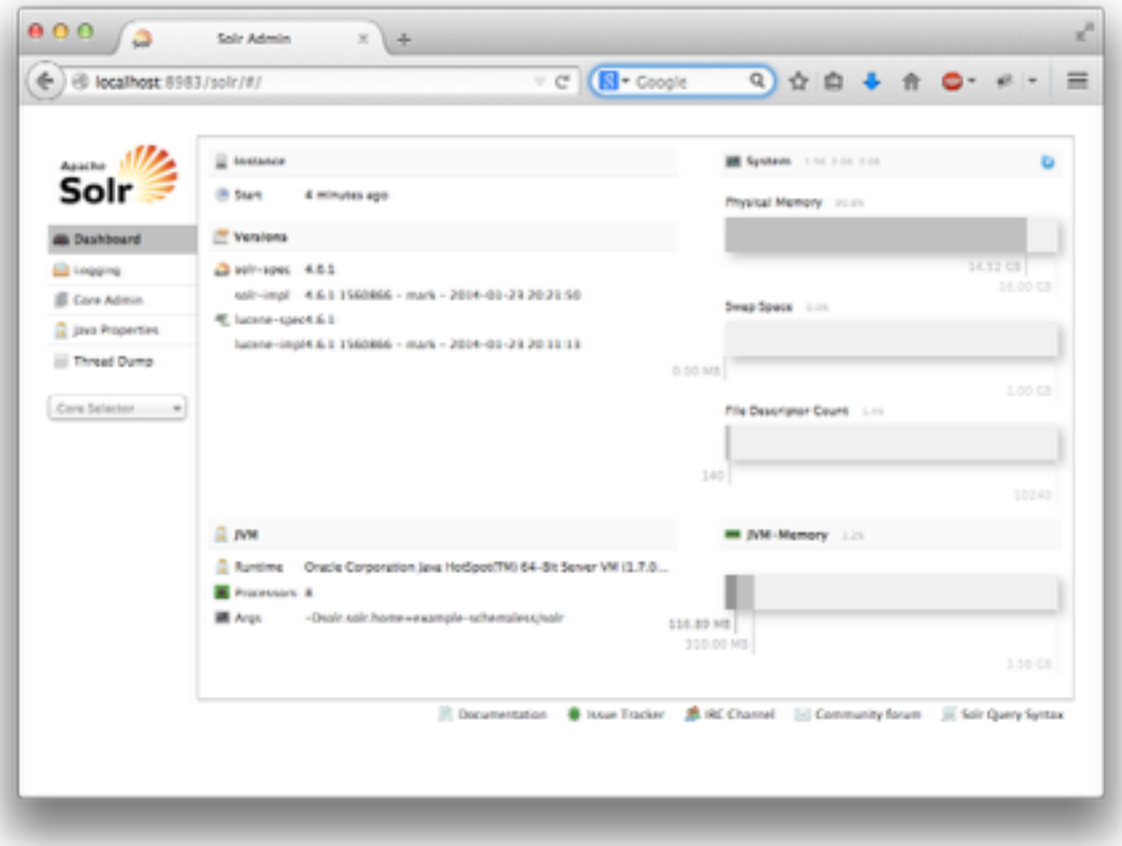
3) Start the solr engine in schema-less mode using the following shell command:

```
java -Dsolr.solr.home=example-schemaless/solr -jar start.jar
```

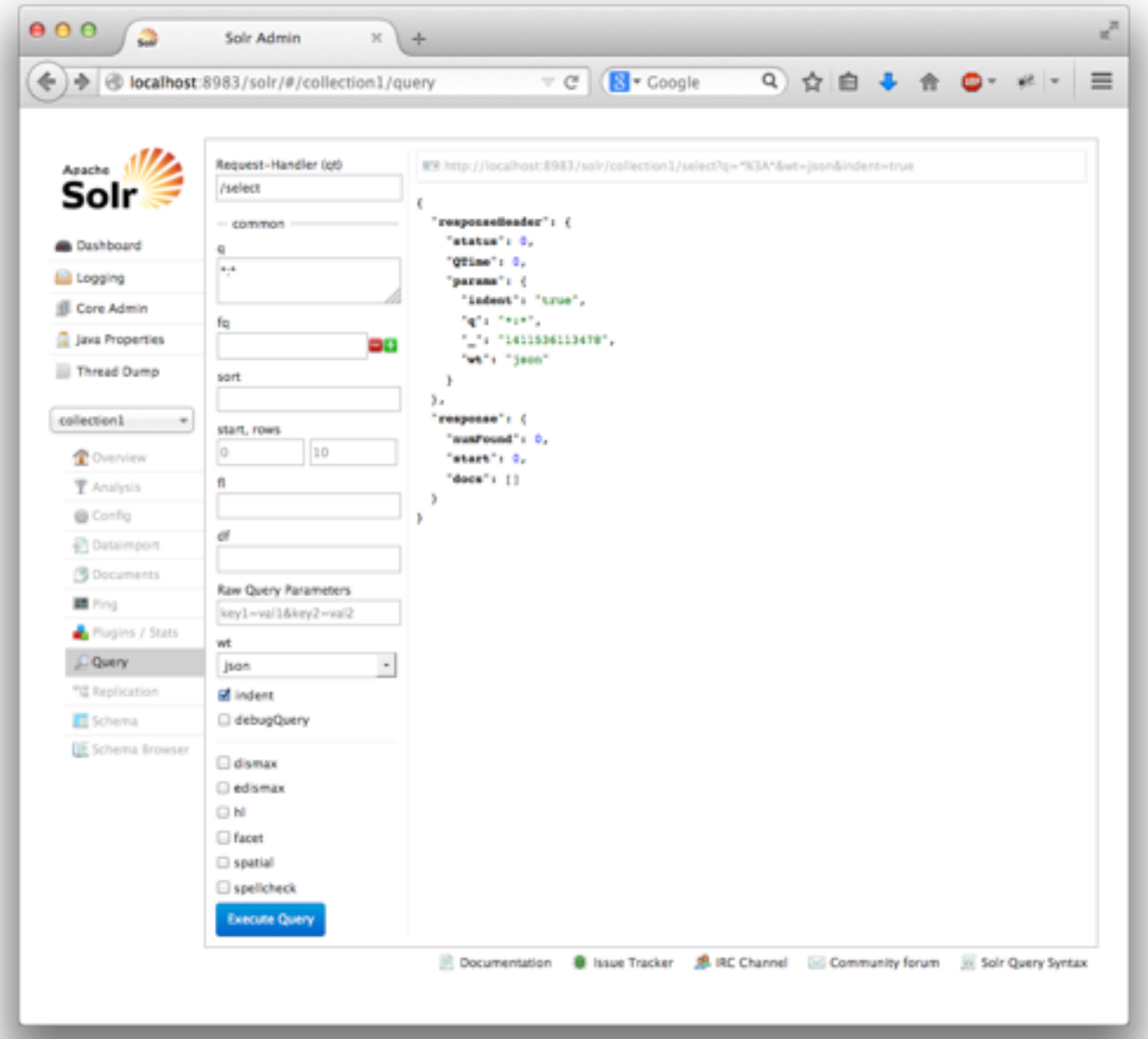
The server should respond with a console log containing:

```
0    [main] INFO  org.eclipse.jetty.server.Server  – jetty-8.1.10.v20130312
  ...
2386 [main] INFO  org.eclipse.jetty.server.AbstractConnector  – Started
SocketConnector@0.0.0.0:8983
```

Solr by default runs on port 8893, and this should now be available via a browser. Enter http://localhost:8983/solr/ to access the admin console:



Use the “core selector” drop down on the left hand column to access the query tool. Clicking on “Execute query” should display an empty result set as seen below:



Note that this also provides the query url at:

```
http://localhost:8983/solr/collection1/select?q=*%3A*&wt=json&indent=true
```

Entering this url in the browser should provide a JSON result set as follows:

```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{
      "indent":"true",
      "q":"*:*",
      "wt":"json"}},
  "response":{"numFound":0,"start":0,"docs":[]
  }}
```

If this is the result, then solr is up and running and ready to go.

Next, open the provided sample xcode project, and locate the fp_constants.m file.

Replace the server ip addresses 10.0.1.23 with the ip address of your local machine:

```
NSString* const kUPDATE_URL         = @"http://10.0.1.23:8983/solr/update?
commit=true";

NSString* const kQUERY_URL          = @"http://10.0.1.23:8983/solr/collection1/
select?wt=json&indent=true&q=";

NSString* const kFACET_QUERY_URL    = @"http://10.0.1.23:8983/solr/collection1/
select?wt=json&df=id&q=*%3A*&fq=" ;
```

You can find your local NAT'ed ip address by entering the following command:

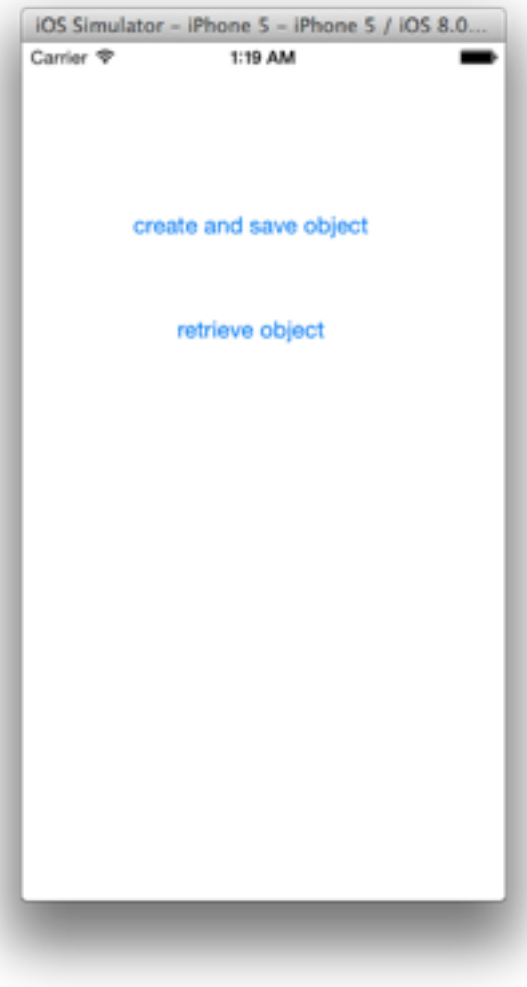
```
ifconfig -a | grep inet
```

It should respond with something like:

```
    inet6 ::1 prefixlen 128
    inet 127.0.0.1 netmask 0xff000000
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet6 fe80::6203:8ff:fe89:9240%en0 prefixlen 64 scopeid 0x4
    inet 10.0.1.23 netmask 0xffffff00 broadcast 10.0.1.255
```

Do not use the loopback address (127…) as this will refer to the device's address when the application is run, even on the simulator. In this example the ip address is 10.0.1.23

Run the Xcode project in the simulator and you should see a screen such as the following:



Click on the “create and save object” button a few times, and then afterwards the “retrieve object” button.

You should be able to see the objects logged to the xcode debug console:

```
2014-09-24 01:31:51.761 FP[4170:566941] Saved object [{"numberOfSongs":
0,"className":"TopBands","id":"gFnpew","hometown":"New York","bandName":"the mud
besiegers","createdAt":"Wed Sep 24 01:31:51 2014"}]

2014-09-24 01:31:51.766 FP[4170:566941] Saved object
[{"cheatMode":false,"className":"GameScore","playerName":"Sean Plott","score":
1337,"id":"kPnnew","createdAt":"Wed Sep 24 01:31:51 2014"}]
2014-09-24 01:31:54.346 FP[4170:566941] object by id query complete <FPObject:
0x79fade10>
```

```
2014-09-24 01:31:54.346 FP[4170:566941] name (
    "Sean Plott"
) score (
    1337
)
2014-09-24 01:31:54.351 FP[4170:566941] multiple objects query complete: found 2
object(s)
2014-09-24 01:31:54.351 FP[4170:566941] object: (
    "Sean Plott"
) - (
    1337
)
2014-09-24 01:31:54.351 FP[4170:566941] object: (
    "Sean Plott"
) - (
    1337
)
2014-09-24 01:31:54.351 FP[4170:566941] multiple objects query complete: found 2
object(s)
2014-09-24 01:31:54.351 FP[4170:566941] object: (
    "Sean Plott"
) - (
    1337
)
2014-09-24 01:31:54.351 FP[4170:566941] object: (
    "Sean Plott"
) - (
    1337
)
2014-09-24 01:31:54.351 FP[4170:566941] multiple objects query complete: found 2
object(s)
2014-09-24 01:31:54.351 FP[4170:566941] object: (
    "the mud besiegers"
) - (
    7
)
2014-09-24 01:31:54.351 FP[4170:566941] object: (
    "the mud besiegers"
) - (
    0
)
```

Returning to the web console url http://localhost:8983/solr/collection1/select?q=*%3A*&wt=json&indent=true in a browser should also display the results in json format:


```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{
      "indent":"true",
      "q":"*:*",
      "wt":"json"}},
  "response":{"numFound":4,"start":0,"docs":[
```

```
    {
      "cheatMode":[false],
      "className":["GameScore"],
      "playerName":["Sean Plott"],
      "score":[1337],
      "id":"AFeOew",
      "createdAt":["Wed Sep 24 01:30:52 2014"],
      "_version_":1480103509445574656},
    {
      "numberOfSongs":[7],
      "className":["TopBands"],
      "id":"cE1OeA",
      "hometown":["New York"],
      "bandName":["the mud besiegers"],
      "createdAt":["Wed Sep 24 01:30:52 2014"],
      "_version_":1480103509448720384},
    {
      "numberOfSongs":[0],
      "className":["TopBands"],
      "id":"gFnpew",
      "hometown":["New York"],
      "bandName":["the mud besiegers"],
      "createdAt":["Wed Sep 24 01:31:51 2014"],
      "_version_":1480103519050530816},
    {
      "cheatMode":[false],
      "className":["GameScore"],
      "playerName":["Sean Plott"],
      "score":[1337],
      "id":"kPnnew",
      "createdAt":["Wed Sep 24 01:31:51 2014"],
      "_version_":1480103519054725120}]
}}
```

Further reading:

Apache Solr

Solr is the popular, blazing fast open source enterprise search platform from the Apache Lucene project. Its major features include powerful full-text search, hit highlighting, faceted search, near real-time indexing, dynamic clustering, database integration, rich document (e.g., Word, PDF) handling, and geospatial search. Solr is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more. Solr powers the search and navigation features of many of the world's largest internet sites.

Solr is written in Java and runs as a standalone full-text search server within a servlet container such as Jetty. Solr uses the Lucene Java search library at its core for full-text indexing and search, and has REST-like HTTP/XML and JSON APIs that make it easy to use from virtually any programming language. Solr's powerful external configuration allows it to be tailored to almost any type of application without Java coding, and it has an extensive plugin architecture when more advanced customization is required.

http://lucene.apache.org/solr/

Scaling Solr

https://cwiki.apache.org/confluence/display/solr/Introduction+to+Scaling+and+Distribution

Solr Cloud

https://cwiki.apache.org/confluence/display/solr/SolrCloud

Integrating Solr with Apache Camel (enterprise service bus framework)

http://camel.apache.org/solr.html

Solr Query API

https://wiki.apache.org/solr/SolrQuerySyntax

Parse iOS API

https://parse.com/docs/ios_guide#top/iOS