JSONStore in iOS applications

Overview

IBM MobileFirst Foundation's **JSONStore** is an optional client-side API providing a lightweight, document-oriented storage system. JSONStore enables persistent storage of **JSON documents**. Documents in an application are available in JSONStore even when the device that is running the application is offline. This persistent, always-available storage can be useful to give users access to documents when, for example, there is no network connection available in the device.

Key features

- Data indexing for efficient searching
- Data encryption in production environments
- Mechanism for tracking local-only changes to the stored data
- Support for multiple users

> **Note:** Some features such as data encryption are beyond the scope of this tutorial. All features are documented in detail in the IBM MobileFirst Foundation user documentation website.

Prerequisite: Make sure the MobileFirst Native SDK was added to the Xcode project. Follow the Adding the MobileFirst Foundation SDK to iOS applications (../../../adding-the-mfpf-sdk/ios/) tutorial.

Jump to:

Adding JSONStore

1. Add the following to the existing `podfile`, located at the root of the Xcode project:

   ```
   pod 'IBMMobileFirstPlatformFoundationJSONStore'
   ```

2. From a **Command-line** window, navigate to the root of the Xcode project and run the command: `pod install` - note that this action may take a while.

Whenever you want to use JSONStore, make sure that you import the JSONStore header:
Objective-C:

```
#import <IBMMobileFirstPlatformFoundationJSONStore/IBMMobileFirstPlatformFoundationJSONStore.h>
```

Swift:

```
import IBMMobileFirstPlatformFoundationJSONStore
```

Basic Usage

Open

Use `openCollections` to open one or more JSONStore collections.

Starting or provisioning a collections means creating the persistent storage that contains the collection and documents, if it does not exists.
If the persistent storage is encrypted and a correct password is passed, the necessary security procedures to make the data accessible are run.

For optional features that you can enable at initialization time, see **Security, Multiple User Support** and **MobileFirst Adapter Integration** in the second part of this tutorial.

```
let collection:JSONStoreCollection = JSONStoreCollection(name: "people")

collection.setSearchField("name", withType: JSONStore_String)
collection.setSearchField("age", withType: JSONStore_Integer)

do {
  try JSONStore.sharedInstance().openCollections([collection], withOptions: nil)
} catch let error as NSError {
  // handle error
}
```

Get

Use `getCollectionWithName` to create an accessor to the collection. You must call `openCollections` before you call `getCollectionWithName`.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)
```

The variable `collection` can now be used to perform operations on the `people` collection such as `add`, `find`, and `replace`.

Add

Use `addData` to store data as documents inside a collection.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

let data = ["name" : "yoel", "age" : 23]

do  {
  try collection.addData([data], andMarkDirty: true, withOptions: nil)
} catch let error as NSError {
  // handle error
}
```

Find

Use `findWithQueryParts` to locate a document inside a collection by using a query. Use `findAllWithOptions` to retrieve all the documents inside a collection. Use `findWithIds` to search by the document unique identifier.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

let options:JSONStoreQueryOptions = JSONStoreQueryOptions()
// returns a maximum of 10 documents, default: returns every document
options.limit = 10

let query:JSONStoreQueryPart = JSONStoreQueryPart()
query.searchField("name", like: "yoel")

do  {
  let results:NSArray = try collection.findWithQueryParts([query], andOptions: options)
} catch let error as NSError {
  // handle error
}
```

Replace

Use `replaceDocuments` to modify documents inside a collection. The field that you use to perform the replacement is `_id,` the document unique identifier.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

var document:Dictionary<String,AnyObject> = Dictionary()
document["name"] = "chevy"
document["age"] = 23

var replacement:Dictionary<String, AnyObject> = Dictionary()
replacement["_id"] = 1
replacement["json"] = document

do {
  try collection.replaceDocuments([replacement], andMarkDirty: true)
} catch let error as NSError {
  // handle error
}
```

This examples assumes that the document `{_id: 1, json: {name: 'yoel', age: 23} }` is in the collection.

Remove

Use `removeWithIds` to delete a document from a collection. Documents are not erased from the collection until you call `markDocumentClean`. For more information, see the **MobileFirst Adapter Integration** section later in this tutorial.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

do {
  try collection.removeWithIds([1], andMarkDirty: true)
} catch let error as NSError {
  // handle error
}
```

Remove Collection

Use `removeCollection` to delete all the documents that are stored inside a collection. This operation is similar to dropping a table in database terms.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

do {
  try collection.removeCollection()
} catch let error as NSError {
  // handle error
}
```

Destroy

Use `destroyData` to remove the following data:

- All documents
- All collections
- All Stores - See **Multiple User Support** later in this tutorial
- All JSONStore metadata and security artifacts - See **Security** later in this tutorial

```
do {
  try JSONStore.sharedInstance().destroyData()
} catch let error as NSError {
  // handle error
}
```

Advanced Usage

Security

You can secure all the collections in a store by passing a `JSONStoreOpenOptions` object with a password to the `openCollections` function. If no password is passed, the documents of all the collections in the store are not encrypted.

Some security metadata is stored in the keychain (iOS).
The store is encrypted with a 256-bit Advanced Encryption Standard (AES) key. All keys are strengthened with Password-Based Key Derivation Function 2 (PBKDF2).

Use `closeAllCollections` to lock access to all the collections until you call `openCollections` again. If

you think of `openCollections` as a login function you can think of `closeAllCollections` as the corresponding logout function.

Use `changeCurrentPassword` to change the password.

```
let collection:JSONStoreCollection = JSONStoreCollection(name: "people")
collection.setSearchField("name", withType: JSONStore_String)
collection.setSearchField("age", withType: JSONStore_Integer)

let options:JSONStoreOpenOptions = JSONStoreOpenOptions()
options.password = "123"

do {
  try JSONStore.sharedInstance().openCollections([collection], withOptions: options)
} catch let error as NSError {
  // handle error
}
```

Multiple User Support

You can create multiple stores that contain different collections in a single MobileFirst application. The `openCollections` function can take an options object with a username. If no username is given, the default username is "jsonstore".

```
let collection:JSONStoreCollection = JSONStoreCollection(name: "people")
collection.setSearchField("name", withType: JSONStore_String)
collection.setSearchField("age", withType: JSONStore_Integer)

let options:JSONStoreOpenOptions = JSONStoreOpenOptions()
options.username = "yoel"

do {
  try JSONStore.sharedInstance().openCollections([collection], withOptions: options)
} catch let error as NSError {
  // handle error
}
```

MobileFirst Adapter Integration

This section assumes that you are familiar with MobileFirst adapters. MobileFirst Adapter Integration is optional and provides ways to send data from a collection to an adapter and get data from an adapter into a collection.

You can achieve these goals by using functions such as `WLResourceRequest`.

Adapter Implementation

Create a MobileFirst adapter and name it "**People**". Define it's procedures `addPerson`, `getPeople`, `pushPeople`, `removePerson`, and `replacePerson`.

```
function getPeople() {
    var data = { peopleList : [{name: 'chevy', age: 23}, {name: 'yoel', age: 23}] };
    WL.Logger.debug('Adapter: people, procedure: getPeople called.');
    WL.Logger.debug('Sending data: ' + JSON.stringify(data));
    return data;
}
function pushPeople(data) {
    WL.Logger.debug('Adapter: people, procedure: pushPeople called.');
    WL.Logger.debug('Got data from JSONStore to ADD: ' + data);
    return;
}
function addPerson(data) {
    WL.Logger.debug('Adapter: people, procedure: addPerson called.');
    WL.Logger.debug('Got data from JSONStore to ADD: ' + data);
    return;
}
function removePerson(data) {
    WL.Logger.debug('Adapter: people, procedure: removePerson called.');
    WL.Logger.debug('Got data from JSONStore to REMOVE: ' + data);
    return;
}
function replacePerson(data) {
    WL.Logger.debug('Adapter: people, procedure: replacePerson called.');
    WL.Logger.debug('Got data from JSONStore to REPLACE: ' + data);
    return;
}
```

Load data from MobileFirst Adapter

To load data from a MobileFirst Adapter use `WLResourceRequest`.

```
// Start - LoadFromAdapter
class LoadFromAdapter: NSObject, WLDelegate {
  func onSuccess(response: WLResponse!) {
    let responsePayload:NSDictionary = response.getResponseJson()
    let people:NSArray = responsePayload.objectForKey("peopleList") as! NSArray
    // handle success
  }

  func onFailure(response: WLFailResponse!) {
    // handle failure
  }
}
// End - LoadFromAdapter

let pull = WLResourceRequest(URL: NSURL(string: "/adapters/People/getPeople"), method: "GET")

let loadDelegate:LoadFromAdapter = LoadFromAdapter()
pull.sendWithDelegate(loadDelegate)
```

Get Push Required (Dirty Documents)

Calling `allDirty` returns and array of so called "dirty documents", which are documents that have local modifications that do not exist on the back-end system.

```
let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

do {
  let dirtyDocs:NSArray = try collection.allDirty()
} catch let error as NSError {
  // handle error
}
```

To prevent JSONStore from marking the documents as "dirty", pass the option `andMarkDirty:false` to `add`, `replace`, and `remove`.

Push changes

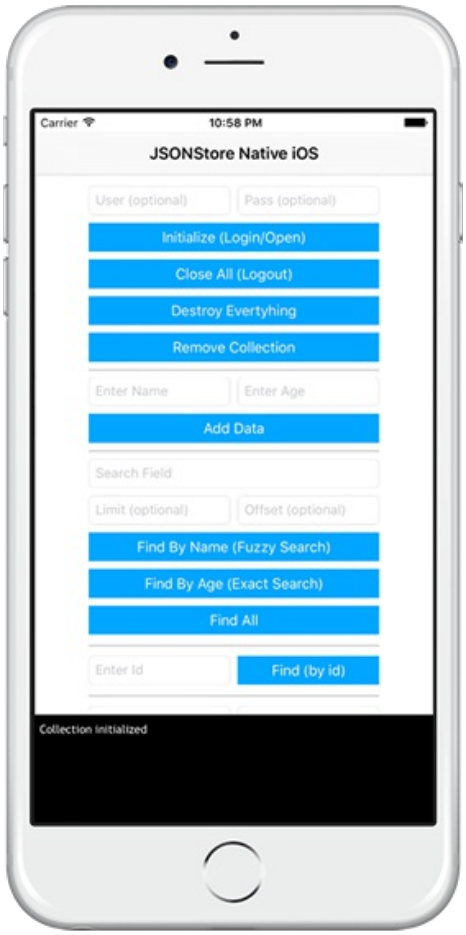
To push changes to a MobileFirst adapter, call the `allDirty` to get a list of documents with modifications and then use `WLResourceRequest`. After the data is sent and a successful response is received make sure you call `markDocumentsClean`.

```
// Start - PushToAdapter
class PushToAdapter: NSObject, WLDelegate {
  func onSuccess(response: WLResponse!) {
    // handle success
  }

  func onFailure(response: WLFailResponse!) {
    // handle failure
  }
}
// End - PushToAdapter

let collectionName:String = "people"
let collection:JSONStoreCollection = JSONStore.sharedInstance().getCollectionWithName(collectionNa
me)

do {
  let dirtyDocs:NSArray = try collection.allDirty()
  let pushData:NSData = NSKeyedArchiver.archivedDataWithRootObject(dirtyDocs)

  let push = WLResourceRequest(URL: NSURL(string: "/adapters/People/pushPeople"), method: "POST
")

  let pushDelegate:PushToAdapter = PushToAdapter()
  push.sendWithData(pushData, delegate: pushDelegate)

} catch let error as NSError {
  // handle error
}
```

Sample application

The JSONStoreSwift project contains a native iOS Swift application that utilizes the JSONStore API set.
Included is a JavaScript adapter Maven project.

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/JSONStoreSwift/tree/release80) the Native iOS project.
Click to download (https://github.com/MobileFirst-Platform-Developer-Center/JSONStoreAdapter/tree/release80) the adapter Maven project.



Sample usage

1. From a **Command-line** window, navigate to the project's root folder and run the command: `mfpdev app register`.
2. The sample uses the `JSONStoreAdapter` contained in the Adapters Maven project. Use either Maven, MobileFirst CLI or your IDE of choice to build and deploy the adapter (../../../adapters/creating-adapters/).
3. Import the project to Xcode, and run the sample by clicking the **Run** button.