# Using JSONStore in Native iOS applications

## Overview

This tutorial is a continuation of the JSONStore Overview tutorial.
The tutorial covers the following topics:

- Adding the JSONStore component
- Basic API Usage
- Advanced Usage
- Sample application
- Additional information

## Adding the JSONStore component

Adding the JSONStore component to native iOS applications is accomplished using CocoaPods. First, make sure the MobileFirst SDK is present by following the instructions in the tutorial: Configuring a Native iOS Application with the MobileFirst Platform SDK (../../../hello-world/configuring-a-native-ios-application-with-the-mfp-sdk/).

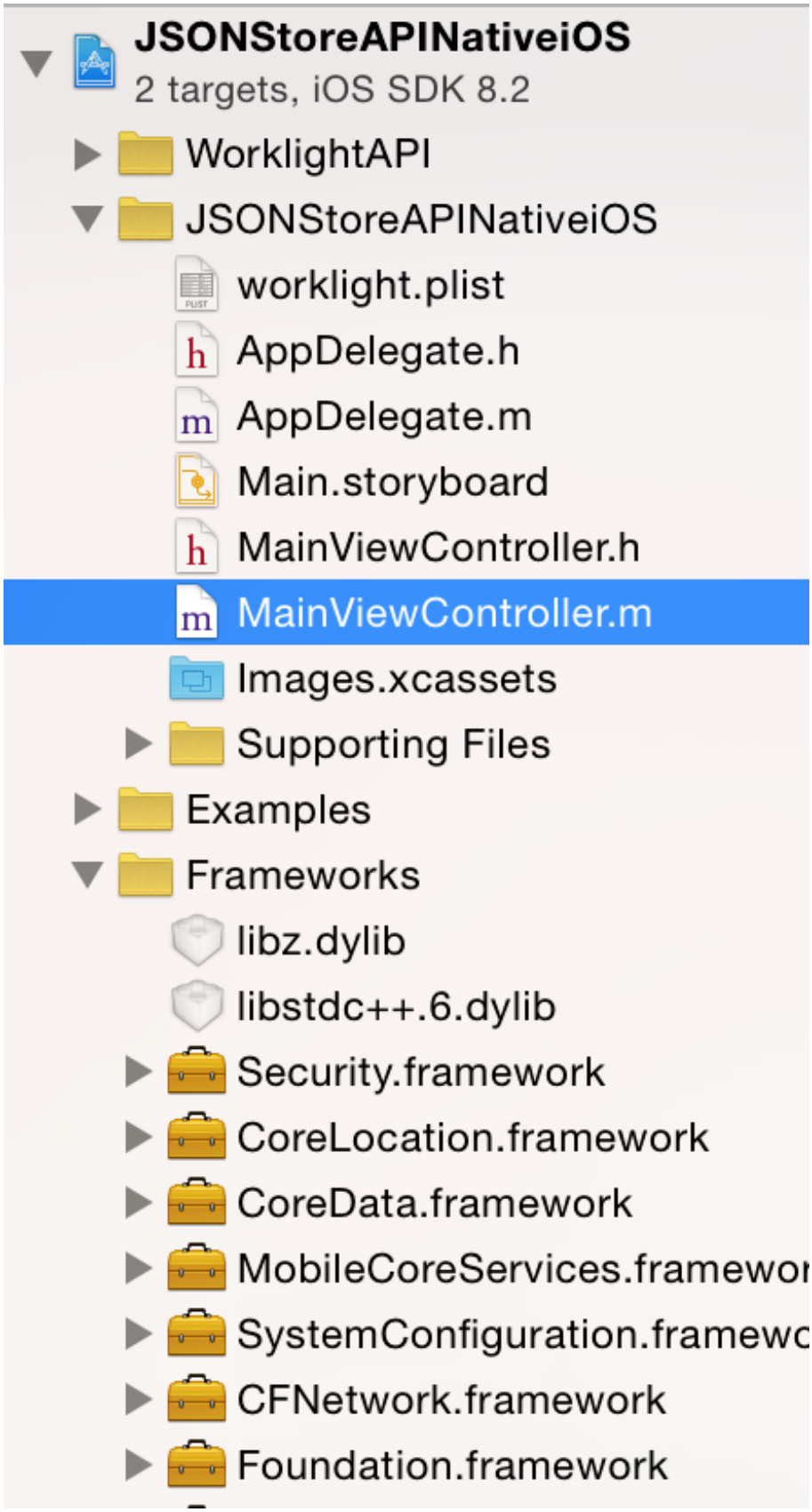
Next, perform the following steps:

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

```
source 'https://github.com/CocoaPods/Specs.git'
pod 'IBMMobileFirstPlatformFoundationJSONStore'
```

3. In **Terminal**, navigate to the root of the Xcode project and run the command: `pod install` - note that this action may take a while.

The JSONStore feature should now be available to you in the Xcode project.




## Basic API 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 module

```
NSError *error = nil;
JSONStoreCollection* collection = [[JSONStoreCollection alloc] initWithName:@"people"];
[collection setSearchField:@"name" withType:JSONStore_String];
[collection setSearchField:@"age" withType:JSONStore_Integer];
[[JSONStore sharedInstance] openCollections:@[collection] withOptions:nil error:@"error"];
```

## Get

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

```
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
```

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

```
NSError *error = nil;
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
NSDictionary *data = @{@"name" : @"yoel", @"age" : @23};
[[collection addData:@[data] andMarkDirty:YES withOptions:nil error:@"error"] intValue];
```

## 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.

```
NSError *error = nil;
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
//Build a query part.
JSONStoreQueryPart *query = [[JSONStoreQueryPart alloc] init];
[query searchField:@"name" like:@"yoel"];
JSONStoreQueryOptions *options = [[JSONStoreQueryOptions alloc] init];
// returns a maximum of 10 documents, default: retuns every document
[options setLimit:@10];
// Count using the query part built above.
NSArray *results = [collection findWithQueryParts:@[query] andOptions:options error:@"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.

```
NSError *error = nil;
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
//Replacing name 'carlos' with name 'carlitos'.
NSDictionary *replacement = @{@"_id": @1, @"json" : @{@"name" : @"chevy", @"age" : @23}};
[collection replaceDocuments:@[replacement] andMarkDirty:YES error:@"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

```
NSError *error = nil;
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];</p>
[collection removeWithIds:@[@1] andMarkDirty:YES error:@"error"];
```

## Remove Collection

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

```
NSError *error = nil;
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
BOOL removeCollectionWorked = [collection removeCollectionWithError:@"error"];
```

### Destroy

Use `destroyDataAndReturnError` 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"

```
NSError *error = nil;
[[JSONStore sharedInstance] destroyDataAndReturnError:@"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 `closeAllCollectionsAndReturnError` 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 `closeAllCollectionsAndReturnError` as the corresponding logout function.

Use `changeCurrentPassword` to change the password.

```
NSError *error = nil;
JSONStoreCollection *collection = [[JSONStoreCollection alloc] initWithName:@"people"];
[collection setSearchField:@"name" withType:JSONStore_String];
[collection setSearchField:@"age" withType:JSONStore_Integer];
JSONStoreOpenOptions *options = [JSONStoreOpenOptions new];
[options setPassword:@"123"];
[[JSONStore sharedInstance] openCollections:@[collection] withOptions:options error:@"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".

```
NSError *error = nil
JSONStoreCollection *collection = [[JSONStoreCollection alloc] initWithName:@"people"];
[collection setSearchField:@"name" withType:JSONStore_String];
[collection setSearchField:@"age" withType:JSONStore_Integer];
JSONStoreOpenOptions *options = [JSONStoreOpenOptions new];
[options setUsername:@"yoel"];
[[JSONStore sharedInstance] openCollections:@[collection] withOptions:options error:@"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 `WLClient invokeProcedure` or your own instance of an `NSURLConnection` if you need more flexibility.

### 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 `WLClient invokeProcedure`.

```
// Start - LoadFromAdapter
@interface LoadFromAdapter : NSObject<WLDelegate>
@end
<@implementation LoadFromAdapter
-(void)onSuccess:(WLResponse *)response {
  NSArray *loadedDocuments = [[response getResponseJson]  objectForKey:@"peopleList"];
  // handle succes
}
-(void)onFailure:(WLFailResponse *)response {
  // handle success
}
@end
// End - LoadFromAdapter
NSError *error = nil;
WLProcedureInvocationData *invocationData = [[WLProcedureInvocationData alloc] initWithAdapterNam
e:@"People" procedureName:@"getPeople"];
LoadFromAdapter *loadDelegate =  [[LoadFromAdapter alloc] init];
WLClient *client = [[WLClient sharedInstance] init];
[client invokeProcedure:invocationData withDelegate:loadDelegate];
```

## Get Push Required (Dirty Documents)

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

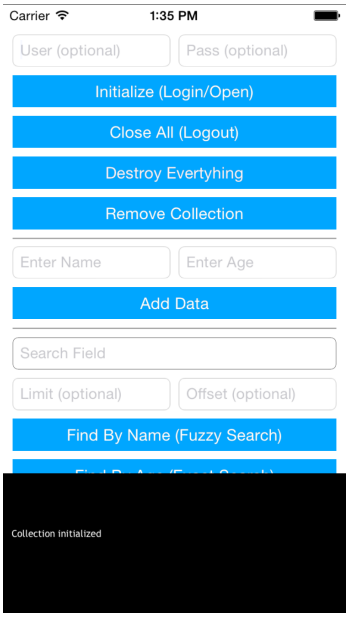
```
NSError* error = nil;
<p>NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
<p>NSArray *dirtyDocs = [collection allDirtyAndReturnError:@"error"];<
```

To prevent JSONStore from marking the documents as "dirty", pass the option `andMarkDirty:NO` to `add`, `replace`, and `remove`

## Push changes

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

```
// Start - PushToAdapter
@interface PushToAdapter :NSObject<WLDelegate>
@end
@implementation PushToAdapter
-(void)onSuccess:(WLResponse *)response {
  // handle success
}
-(void)onFailure:(WLFailResponse *)response {
  // handle faiure
}
@end
// End - PushToAdapter
NSError* error = nil;
NSString *collectionName = @"people";
JSONStoreCollection *collection = [[JSONStore sharedInstance] getCollectionWithName:collectionName
];
NSArray *dirtyDocs = [collection allDirtyAndReturnError:@"error"];
WLProcedureInvocationData *invocationData = [[WLProcedureInvocationData alloc] initWithAdapterNam
e:@"People" procedureName:@"pushPeople"];
[invocationData setParameters:@[dirtyDocs]];
PushToAdapter *pushDelegate =  [[PushToAdapter alloc] init];
WLClient *client = [[WLClient sharedInstance] init];
[client invokeProcedure:invocationData withDelegate:pushDelegate];
```




## Sample application

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/JSONStore/tree/release71) the MobileFirst project.

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/JSONStoreObjC/tree/release71) the Native project.

The Native iOS project contains an application that demonstrates the use of JSONStore.

## Additional information

For more information about JSONStore, see the product user documentation.