iOS - Cloudant NoSQL DB API

Overview

This tutorial shows how to apply technology available in IBM MobileFirst Platform Foundation to store data for iOS mobile applications. By using Cloudant NoSQL DB APIs, you can complete specific database operations, such as creating, reading, updating, deleting, and synchronizing data between local and remote databases. This tutorial provides a basic overview of these APIs and explains how to configure and run the sample, in the following topics:

- Using the IMFData SDK
- Creating local data stores
- Creating remote data stores
- Setting user permissions for remote stores
- Modeling data for iOS applications
- Creating and updating data
- Supporting synchronization and offline storage
- Running the iOS sample
- Obtaining and configuring the required SDKs
- Enabling encryption
- BlueList application flow

Using the IMFData SDK

After the IMFData SDK is installed, you can begin to initialize and use the SDK in your native iOS application.

UPDATE TO SWIFT

```
// Initialize the IMFDataManager
IMFDataManager *manager = [IMFDataManager initializeWithUrl:cloudantProxyUrl];
```

The value for `cloudantProxyUrl` is the URL of the MobileFirst Data Proxy as it was configured during installation. The URL includes the IP address, host, and context root that you defined.
For example: `http://localhost:9080/imf-data-proxy`
In the native iOS sample that comes with this tutorial, the value for the `cloudantProxyUrl` is set in the `bluelist.plist` file.

Creating local data stores

You can create a Store object to allow your application to access a local database, which can be used even when the application is offline.

UPDATE TO SWIFT

```
//Get reference to IMFDataManager
IMFDataManager *manager = [IMFDataManager sharedInstance];
NSString *dbname=@"todosdb";
NSError *error = nil;

//Create local store
CDTStore *datastore = [manager localStore:dbname error:&error];
if (error) {
        [NSException raise:@"DBCreationFailure" format: @"Could not create DB with name %@", dbname]
;
} else {
        NSLog(@"Local data store created successfully");
}
```

Creating remote data stores

You can also create a Store object to allow your application to access a remote database.

UPDATE TO SWIFT

```
// Get reference to data manager
IMFDataManager *manager = [IMFDataManager sharedInstance];
NSString *dbname = @"todosdb";

// Create remote store
[manager remoteStore:dbname completionHandler:^(CDTStore *store, NSError *error) {
    if(error){
        // Handle error
    }else{
        CDTStore *remotedatastore = store;
        NSLog(@"Successfully created store);
    }
}];
```

Setting user permissions for remote stores

You can set specific permissions for users to access remote stores.

UPDATE TO SWIFT

```
[manager setCurrentUserPermissions: DB_ACCESS_GROUP_MEMBERS forStoreName: @"todosdb" com
pletionHander:^(BOOL success, NSError *error) {
    if(error){
        // Handle error
    }else{
        // setting permissions was successful
    }
}];
```

UPDATE LINK TO 8.0 DOC'S **Note:** In the sample, the user is authenticated via OAuth. OAuth has been configured through adapter-based authentication. You can find further instructions on setting up access with OAuth capabilities and the MobileFirst Data Proxy in the configuring OAuth security documentation

(http://ibm.biz/knowctr#SSHS8R_7.0.0/com.ibm.worklight.dev.doc/cloud/data/t_data_cloudantsec.html%23oauth?lang=en).

Modeling data for iOS applications

In iOS applications, you can use the `CDTDataObjectMapper` class to map native objects to the JSON document format. When you create a data store with the `IMFDataManager` API, a `CDTDataObjectMapper` is created automatically and is set on the `CDTStore` object. In the sample, a custom `TodoItem` class enables you to store custom data as objects in the application. When you create a custom `CDTDataObjectMapper` class, make sure that it meets the following requirements:

- Conform to the `IMFDataObject` protocol.
- Have the `IMFDataObject` protocol metadata property set on the class interface.
- Extend `NSObject`.

TodoItem implementation (`TodoItem.h`)

UPDATE TO SWIFT

```
@interface TodoItem : NSObject <CDTDataObject>
@property NSString *name;
@property NSNumber *priority;
//Required by the IMFDataObject protocol
@property (strong, nonatomic, readwrite) CDTDataObjectMetadata *metadata;
@end
```

You must then register the class and data type with the `CDTDataObjectMapper`. In the sample, the `TableViewController` does that after the store is created.

UPDATE TO SWIFT

```
//using the existing store
[self.datastore.mapper setDataType:@"TodoItem" forClassName:NSStringFromClass([TodoItem class])];
```

Creating and updating data

Using the same operation, you can save new objects and save changes to existing objects in a data store.

Creating Todo Items

UPDATE TO SWIFT

```
//using a store that was created previously
- (void) createItem: (TodoItem*) item
{
    //save will perform a create because the item object does not exist yet in the DB.
    [self.datastore save:item completionHandler:^(NSObject *object, NSError *error) {
        if (error) {
            NSLog(@"createItem failed with error: %@", error);
        } else {
            [self listItems:nil];
        }
    }];
}
```

Updating Todo Items

UPDATE TO SWIFT

```
//using a store that was created previously
- (void) updateItem: (TodoItem*) item
{
    //save will perform a create because the CDTDocumentRevision already exists.
    [self.datastore save:item completionHandler:^(NSObject *object, NSError *error) {
        if (error) {
            NSLog(@"updateItem failed with error: %@", error);
        } else {
            [self listItems:nil];
        }
    }];
}
```

Deleting data

To delete an object in a data store, pass the object to the `delete: completionHandler` method:

UPDATE TO SWIFT

```
//using a store that was created previously
-(void) deleteItem: (TodoItem*) item
{
    [self.datastore delete:item completionHandler:^(NSString *deletedObjectId, NSString *deletedRevisionId,
NSError *error) {
        if (error != nil) {
            NSLog(@"deleteItem failed with error: %@", error);
        } else {
            [self listItems:nil];
        }
    }];
}
```

Querying data

UPDATE LINK TO 8.0 DOC'S You can query for objects that have an object mapper. The Cloudant query API provides convenient methods for querying with `NSPredicate` and for querying by data type. For more information about these functions, see Querying data (http://ibm.biz/knowctr#SSHS8R_7.0.0/com.ibm.worklight.dev.doc/cloud/data/t_data_sdk_query.html) in the user documentation.

Supporting synchronization and offline storage

By using the data manager API, you can synchronize data between local storage on the device and remote stored instances.

Pull replication

When pull replication runs, the local database within the mobile device is updated with what exists in the remote database.

UPDATE TO SWIFT

```
- (void)pullItems
{
    // store is an existing CDTStore object created using IMFDataManager remoteStore
    NSError *replicationError;
    CDTPullReplication *pull = [manager pullReplicationForStore: store.name];
    CDTReplicator *replicator = [manager.replicatorFactory oneWay:pull error:&replicationError]
;
    if(replicationError){
        // Handle error
    }else{
        // replicator creation was successful
    }
    [replicator startWithError:&replicationError];
    if(replicationError){
        // Handle error
    }else{
        // replicator start was successful
    }
}
```

Push replication

When push replication runs, the data from the local database within the mobile device is sent to the remote database.

UPDATE TO SWIFT

```
- (void)pushItems
{
    // store is an existing CDTStore object created using IMFDataManager localStore
    NSError *replicationError;
    CDTPushReplication *push = [manager pushReplicationForStore: store.name];
    CDTReplicator *replicator = [manager.replicatorFactory oneWay:push error:&replicationError]
;
    if(replicationError){
        // Handle error
    }else{
        // replicator creation was successful
    }
    [replicator startWithError:&replicationError];
    if(replicationError){
        // Handle error
    }else{
        // replicator start was successful
    }
}
```

Note: The code snippets above might be different from the code that you can see in the sample. These snippets have been created to give the clearest and most understandable outline of the APIs and how they are used. For the sample to be easier to use, all the implementation for the above operations can be found in the `TableViewController`.

Running the iOS sample

Make sure that you have properly configured and started the following instances:

MobileFirst Server with `CloudantAuthenticationAdapter` and `iOSBlueList` Native API deployed Cloudant Data Local Layer Edition MobileFirst Data Proxy Server configured against MobileFirst Server and the Cloudant Data Layer Local Edition

If you have not completed some of these tasks, review the proper setup section of the Working with Cloudant NoSQL DB API (../#overview) tutorial.

Obtaining and configuring the required SDKs

1. Make sure that you have correctly installed and set up CocoaPods. If you have not done so, complete the following:
    1. Install CocoaPods by entering the following terminal command:

        ```
        $ sudo gem install cocoapods
        ```

    2. Set up CocoaPods by entering the following terminal command:

        ```
        $ pod setup
        ```

        The next step is to download and install the required dependencies for this project by using the provided `Podfile`. If you want to create or have your own `podfile`, include the following content:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '7.0'
pod 'IMFDataLocal', '1.0.0'
```

2. To install the dependencies as defined in a `podfile`, navigate to the Xcode project directory in terminal (in the sample: `/BlueList-On-Premise/iOS/objective-c/`) and run the following command:

   ```
   $ pod install
   ```

   Note: For the `pod install` command to work correctly, your `podfile` must be named `Podfile` or `podfile`.
3. After the dependencies are installed, open the `bluelist-objective-c.xcworkspace` file. When using CocoaPods, you must use the `.xcworkspace` file instead of the `.xcodeproject` file because of how the dependencies are configured. This pod installation will also pull in the `IBMMobileFirstPlatformFoundation` framework as it is a required dependency of `IMFDataLocal`.
4. After the project is open, check and update the following items:

- In the `bluelist.plist` file, set the `cloudantProxyUrl` to your MobileFirst Data Proxy Server location.
- In the `worklight.plist` file, make sure to check all the values and update them to match the MobileFirst Server instance that you have deployed.

Now you can run the sample on the simulators that are provided by Xcode or on a supported iOS device.

Enabling encryption

It is possible to encrypt the local data stores in order to secure data that is being stored on the device.

Prerequisite: To use the encryption functionality as explained here, you must have the `IMFDataLocal/SQLCipher` pod installed.

To begin the process of encrypting the local databases, you must first edit the `podfile` to include the `IMFDataLocal/SQLCipher` pod. This pod replaces the IMFDataLocal reference in the current `podfile`. Here is an example of the modified `podfile`:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '7.0'
pod 'IMFDataLocal/SQLCipher'
```

A `pod install` must be completed after these changes have been made in order to have the correct encryption dependencies configured. To accomplish this, navigate to the Xcode project directory in terminal (in the sample: `/BlueList-On-Premise/iOS/objective-c/`) and run the following command:

```
$ pod install
```

Now the project has the dependencies that are required to encrypt the local databases correctly. To encrypt these databases in your client-side code, first create an `CDTEncryptionKeyProvider` and then create a local data store with this `keyProvider`:

UPDATE TO SWIFT

```
//Initialize a key provider
id<CDTEncryptionKeyProvider> keyProvider = [CDTEncryptionKeychainProvider providerWithPassword:@
"password" forIdentifier:@"user"];

//Initialize a local store
self.datastore = [manager localStore:dbname withEncryptionKeyProvider:keyProvider error:&dbCreateError
];
```

You must also use the `CDTEncryptionKeyProvider` that you defined when you created `CDTPullReplication` and `CDTPushReplication`:

UPDATE TO SWIFT

```
//pull replication
self.pullReplication   = [[IMFDataManager sharedInstance] pullReplicationForStore:dbname withEncryption
KeyProvider:keyProvider];

//push replication
self.pushReplication   = [[IMFDataManager sharedInstance] pushReplicationForStore:dbname withEncrypti
onKeyProvider:keyProvider];
```

In the sample application, the encryption code is already provided in the `TableViewController`. By default, encryption is not enabled until an `encryptionPassword` is provided in the `bluelist.plist` file. After `encryptionPassword` is configured, the application uses this password to encrypt the local data store by using the above mechanisms. If `encryptionPassword` is left blank in the `bluelist.plist` file, encryption does not occur.

BlueList application flow

When the application starts, the `AppDelegate` instance initializes the connection to MobileFirst Server. After initialization, the application creates and registers the `BlueListChallangeHandler` handler.

UPDATE TO SWIFT

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOpti
ons {
    WLClient *wlClient = [WLClient sharedInstance];

    // Register the Challenge Handler
    BlueListChallengeHandler *challengeHandler = [[BlueListChallengeHandler alloc] init];
    [wlClient registerChallengeHandler: challengeHandler];

    return YES;
}
```

The sample silently authenticates a user by calling `CloudantAutheticationAdapter` with the correct username and password parameters. This is completed in the `BlueListChallengeHandler`.

-(void) handleChallenge:(WLResponse *)response { NSLog(@"Inside handleChallenge - silently loggin in"); WLProcedureInvocationData *invocationData = [[WLProcedureInvocationData alloc] initWithAdapterName:@"CloudantAuthenticationAdapter" procedureName:@"submitAuthentication"]; invocationData.parameters = @[@"james", @"42"];

```
[self submitAdapterAuthentication: invocationData options: nil];
```

}

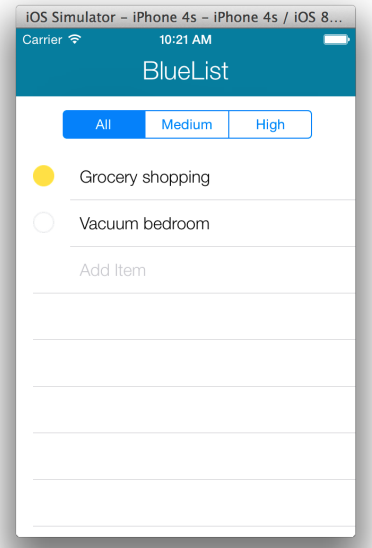
After authentication, the database is configured in the `setupIMFDatabase` method in the `TableViewController`. This database configuration consists of the following steps:

1. Initializing the SDK
2. Creating the local and remote databases.
3. Setting permissions for the user that was created.
4. Setting the `replicatorFactory`, `pullReplication` and `pushReplication`.
5. Registering the `TodoItem` class and data type with the `CDTDataObjectMapper`.

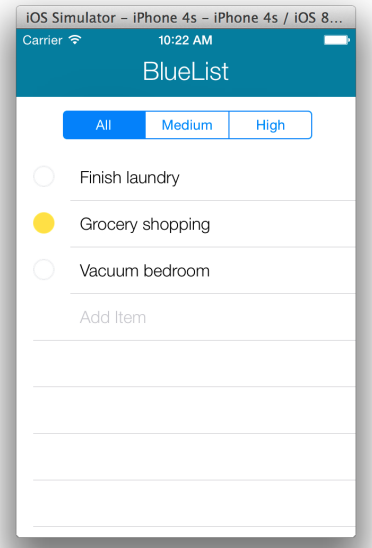
Note: For simplicity, the name of the database that gets created is currently set to the variable `IBM_DB_NAME` in the `TableViewController`. By design, the `setupIMFDatabase` method allows you to pass any database name as a parameter.

The `TableViewController` class provides functions for creating, updating, deleting, and listing items, and functions for push and pull replication.

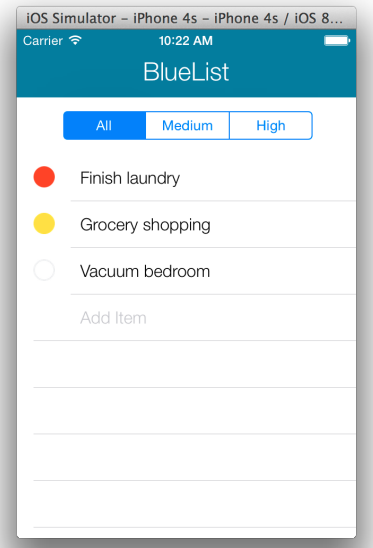
After completing initialization, the application does an initial pull from the remote database. The user can now add, delete, and modify list items in the application, and update the corresponding priority (red = high, yellow = medium, white = low). By design, the application starts synchronization only when the user manually pulls down the list. On pull down, the application first completes a pull replication from the remote database to the local. After this operation has completed, push replication occurs, which sends any updated local data to the remote database. The local and remote databases are then synchronized.



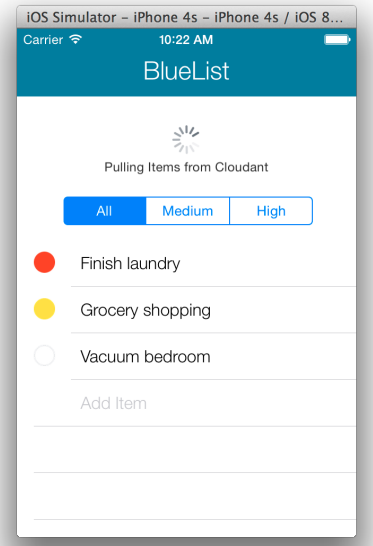
Initial List



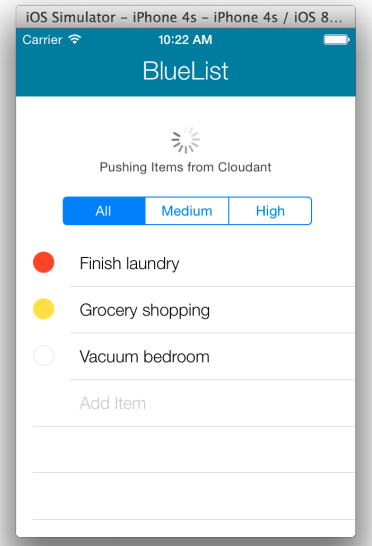
Adding an Item



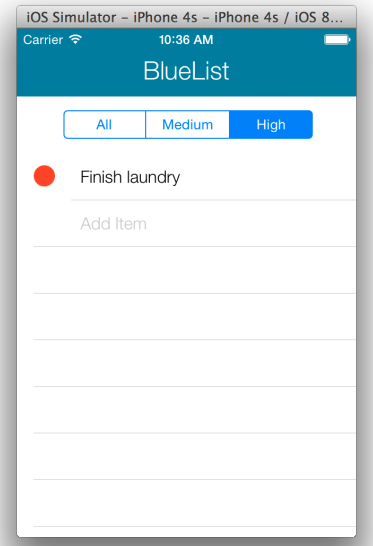
Updating Priority



Pull Replication



Push Replication



Check Priority List