Migrating apps storing mobile data in Cloudant with IMFData or Cloudant SDK

Overview

You can store data for your mobile application in a Cloudant database. Cloudant is an advanced NoSQL database that can handle a wide variety of data types, such as JSON, full-text, and geospatial data. The SDK is available for Java™ , Objective-C, and Swift.

> CloudantToolkit and IMFData frameworks are discontinued in IBM MobileFirst Foundation v8.0.

- For iOS, use the CDTDatastore (https://github.com/cloudant/CDTDatastore) SDK as a replacement for CloudantToolkit and IMFData frameworks.
- For Android, Use the Cloudant Sync Android SDK (https://github.com/cloudant/sync-android) as a replacement for CloudantToolkit and IMFData frameworks. With Cloudant Sync, you can persist data locally and replicate with a remote data store.

If you want to access remote stores directly, use REST calls in your application and refer to the Cloudant API Reference (https://docs.cloudant.com/api.html).

Cloudant versus JSONStore

You might consider using JSONStore instead of Cloudant in the following scenarios:

- When you are storing data on the mobile device that must be stored in a FIPS 140-2 compliant manner.
- When you need to synchronize data between the device and the enterprise.
- When you are developing a hybrid application.

For more information about JSONStore, see JSONStore (../../application-development/jsonstore).

Jump to

Integrating MobileFirst and Cloudant security

Adapter sample

To download the sample, see Sample: mfp-bluelist-on-premises (https://github.com/MobileFirst-Platform-Developer-Center/BlueList-On-Premise).

To understand the MobileFirst adapter that is included with the Bluelist sample, you must understand both Cloudant security (https://cloudant.com/for-developers/faq/auth/) and MobileFirst security framework (../../authentication-and-security).

The Bluelist adapter sample has two primary functions:

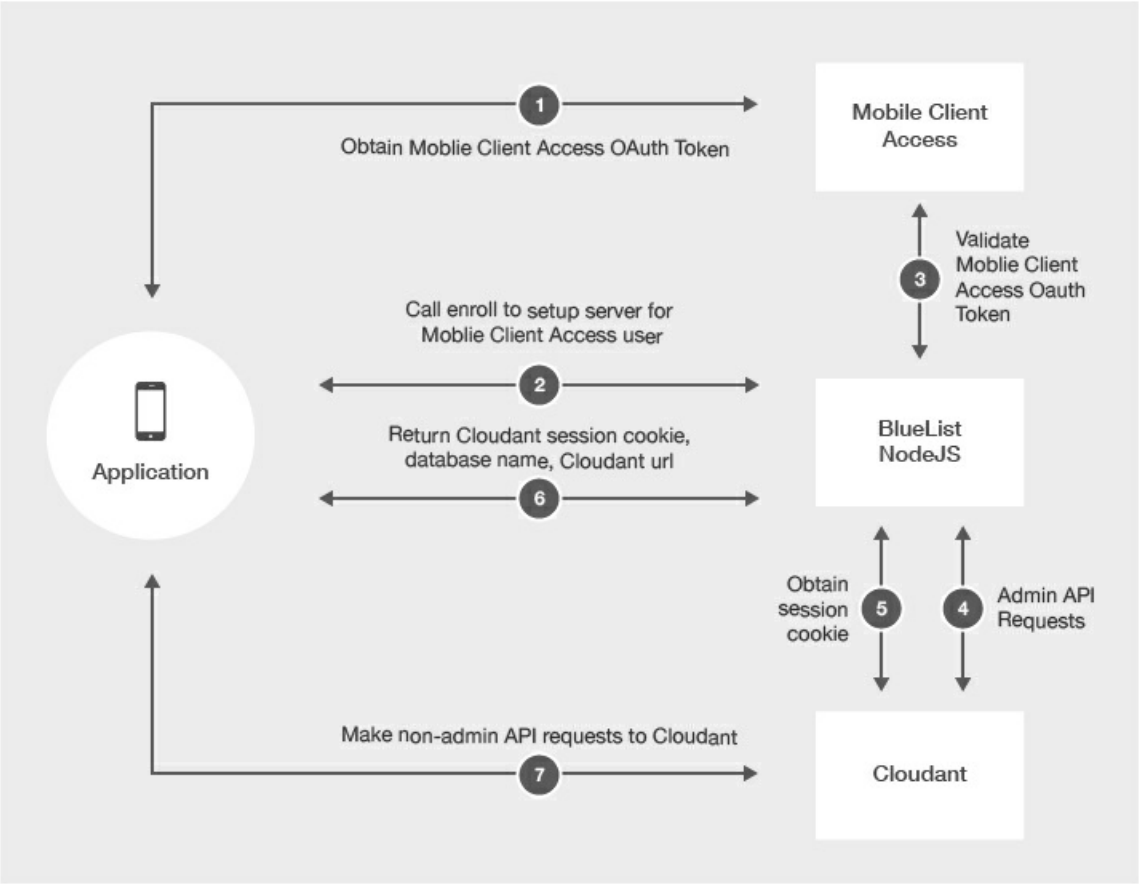
- Exchange MobileFirst OAuth tokens for Cloudant session cookies
- Perform the required admin requests to Cloudant from the Bluelist sample.

The sample demonstrates how to perform API requests that require admin access on the server where it is secure. While it is possible to place your admin credentials on the mobile device, it is a better practice to restrict access from mobile devices.

The Bluelist sample integrates MobileFirst security with Cloudant security. The MobileFirst adapter sample maps a MobileFirst identity to a Cloudant identity. The mobile device receives a Cloudant session cookie to perform non-admin API requests. The sample uses the Couch Security model.

Enroll REST endpoint

The following diagram illustrates the integration performed by the Bluelist adapter sample /**enroll** endpoint.



1. Mobile device obtains the MobileFirst OAuth token from the MobileFirst Server.
2. Mobile device calls the /**enroll** endpoint on the MobileFirst adapter.
3. The MobileFirst adapter sample validates the MobileFirst OAuth token with the MobileFirst Server.
4. If valid, performs admin API requests to Cloudant . The sample checks for an existing Cloudant user in the **_users** database.
    - If the user exists, look up Cloudant user credentials in the **_users** database.
    - If a new user is passed, use the Cloudant admin credentials, create a new Cloudant user and store in the **_users** database.
    - Generate a unique database name for the user and create a remote database on Cloudant with that name.
    - Give the Cloudant user permissions to read/write the newly created database.
    - Create the required indexes for the Bluelist application.
5. Request a new Cloudant session cookie.
6. The MobileFirst adapter sample returns a Cloudant session cookie, remote database name, and Cloudant URL to the mobile device.
7. Mobile device makes requests directly to Cloudant until the session cookie expires.

sessioncookie REST Endpoint

In the case of an expired session cookie, the mobile device can exchange a valid MobileFirst OAuth token for a Cloudant session cookie with the /**sessioncookie** endpoint.

Creating databases

Accessing local data stores

You can use a local data store to store data on the client device for fast access, even when offline.
To create Store objects to access a local database, supply a name for the data store.

> **Important:** The database name must be in lowercase.

iOS

BEFORE (with IMFData/CloudantToolkit):

Objective-C

```
//Get reference to data manager
IMFDataManager *manager = [IMFDataManager sharedInstance];
NSString *name = @"automobiledb";
NSError *error = nil;
```

Swift

```
//Create local store
CDTStore *store = [manager localStore:name error:&error];
let manager = IMFDataManager.sharedInstance()
let name = "automobiledb"

var store:CDTStore?
do {
   store = try manager.localStore(name)
} catch let error as NSError {
   // Handle error
}
```

AFTER (with Cloudant Sync):

Objective-C

```
// Get reference to datastore manager
CDTDatastoreManager *datastoreManager = existingDatastoreManager;
NSString *name = @"automobiledb";
NSError *error = nil;

//Create datastore
CDTDatastore *datastore = [datastoreManager datastoreNamed:name error:&error];
```

Swift

```
// Get reference to datastore manager
let datastoreManager:CDTDatastoreManager = existingDatastoreManager
let name:String  = "automobiledb"

//Create local store
var datastore:CDTDatastore?
do{
   datastore = try datastoreManager.datastoreNamed(name)
}catch let error as NSError{
   // Handle error
}
```

Android

BEFORE (with IMFData/CloudantToolkit):

```
// Get reference to DataManager
DataManager manager = DataManager.getInstance();

// Create local store
String name = "automobiledb";

Task<Store> storeTask = manager.localStore(name);
storeTask.continueWith(new Continuation<Store, Void>() {
   @Override
   public Void then(Task<Store> task) throws Exception {
      if(task.isFaulted()){
         // Handle error
      }else{
         // Do something with Store
         Store store = task.getResult();
      }
      return null;
   }
});
```

AFTER (with Cloudant Sync):

```
// Create DatastoreManager
    File path = context.getDir("databasedir", Context.MODE_PRIVATE);
    DatastoreManager manager = new DatastoreManager(path.getAbsolutePath());

    // Create a Datastore
    String name = "automobiledb";
    Datastore datastore = manager.openDatastore(name);
```

Creating remote data stores

To save data in the remote store, supply the data store name.

iOS

BEFORE (with IMFData/CloudantToolkit):

Objective-c

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

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

Swift

```
let manager = IMFDataManager.sharedInstance()
let name = "automobiledb"

manager.remoteStore(name, completionHandler: { (createdStore:CDTStore!, error:NSError!) -> Void in
  if nil != error {
    //Handle error
  } else {
    let store:CDTStore = createdStore
    print("Successfully created store: \(store.name)")
  }
})
```

AFTER (with Cloudant Sync):

Objective-c

```

```

Swift

```

```

Android

BEFORE (with IMFData/CloudantToolkit):

```

```

AFTER (with Cloudant Sync):

```

```

Encrypting data on the device

To enable the encryption of local data stores on mobile devices, you must make updates to your application to include encryption capabilities and create encrypted data stores.

Encrypting data on iOS devices

1. Obtain the encryption capabilities with CocoaPods.
   - Open your Podfile and add the following line:

     Before (with IMFData/CloudantToolkit):

     ```
     pod 'IMFDataLocal/SQLCipher'
     ```

     After (with Cloudant Sync):

     ```
     pod 'CDTDatastore/SQLCipher'
     ```

     For more information, see the CDTDatastore encryption documentation (https://github.com/cloudant/CDTDatastore/blob/master/doc/encryption.md).
   - Run the following command to add the dependencies to your application.

     ```
     pod install
     ```

2. To use the encryption feature within a Swift application, add the following imports to the associated bridging header for the application:

   Before (with IMFData/CloudantToolkit):

   ```
   #import <CloudantSync.h>
   #import <CloudantSyncEncryption.h>
   #import <CloudantToolkit/CloudantToolkit.h>
   #import <IMFData/IMFData.h>
   ```

   After (with Cloudant Sync):

   ```
   #import <CloudantSync.h>
   #import <CloudantSyncEncryption.h>
   ```

3. Initialize your local store for encryption with a key provider.

   > **Warning:** If you change the password after creating the database, an error occurs because the existing database cannot be decrypted. You cannot change your password after the database has been encrypted. You must delete the database to change passwords.

   BEFORE (with IMFData/CloudantToolkit):

   Objective-C

   ```
   //Get reference to data manager
   IMFDataManager *manager = [IMFDataManager sharedInstance];
   NSString *name = @"automobiledb";
   NSError *error = nil;

   // Initalize a key provider
   id<CDTEncryptionKeyProvider> keyProvider = [CDTEncryptionKeychainProvider providerWithPassword: @"passw0rd" forIdentifier:
   @"identifier"];

   //Initialize local store
   CDTStore *localStore = [manager localStore: name withEncryptionKeyProvider: keyProvider error: &error];
   ```

   Swift

   ```
   let manager = IMFDataManager.sharedInstance()
   let name = "automobiledb"

   let keyProvider = CDTEncryptionKeychainProvider(password: "passw0rd", forIdentifier: "identifier")
   var store:CDTStore?
   do {
      store = try manager.localStore(name, withEncryptionKeyProvider: keyProvider)
   } catch let error as NSError {
      // Handle error
   }
   ```

   AFTER (with Cloudant Sync):

Objective-C

```
// Get reference to datastore manager
CDTDatastoreManager *datastoreManager = existingDatastoreManager;
NSString *name = @"automobiledb";
NSError *error = nil;

// Create KeyProvider
id<CDTEncryptionKeyProvider> keyProvider = [CDTEncryptionKeychainProvider providerWithPassword: @"passw0rd" forIdentifier:
@"identifier"];

//Create local store
CDTDatastore *datastore = [datastoreManager datastoreNamed:name withEncryptionKeyProvider:keyProvider error:&error];
```

Swift

```
// Get reference to datastore manager
let datastoreManager:CDTDatastoreManager = existingDatastoreManager
let name:String  = "automobiledb"

//Create local store
var datastore:CDTDatastore?
let keyProvider = CDTEncryptionKeychainProvider(password: "passw0rd", forIdentifier: "identifier")
do{
   datastore = try datastoreManager.datastoreNamed(name, withEncryptionKeyProvider: keyProvider)
}catch let error as NSError{
   // Handle error
}
```

4. When you are replicating data with an encrypted local store, you must initialize the CDTPullReplication and CDTPushReplication methods with a key provider.

BEFORE (with IMFData/CloudantToolkit):

Objective-C

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

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

// pull replication
CDTPullReplication *pull = [manager pullReplicationForStore: databaseName withEncryptionKeyProvider: keyProvider];

// push replication
CDTPushReplication *push = [manager pushReplicationForStore: databaseName withEncryptionKeyProvider: keyProvider];
```

Swift

```
//Get reference to data manager
let manager = IMFDataManager.sharedInstance()
let databaseName = "automobiledb"

// Initalize a key provider
let keyProvider = CDTEncryptionKeychainProvider(password: "password", forIdentifier: "identifier")

// pull replication
let pull:CDTPullReplication = manager.pullReplicationForStore(databaseName, withEncryptionKeyProvider: keyProvider)

// push replication
let push:CDTPushReplication = manager.pushReplicationForStore(databaseName, withEncryptionKeyProvider: keyProvider)
```

AFTER (with Cloudant Sync):

Replication with an encrypted database requires no changes from replication with an unencrypted database.

Encrypting data on Android devices

To encrypt data on an Android device, obtain encryption capabilities by including the correct libraries in your application. Then, you can initialize your local store for encryption and replicate data.

1. Add the Cloudant Toolkit library as a dependency in your build.gradle file:

   BEFORE (with IMFData/CloudantToolkit):

```
repositories {
mavenCentral()
}

dependencies {
   compile 'com.ibm.mobile.services:cloudant-toolkit-local:1.0.0'
}
```

   AFTER (with Cloudant Sync):

```
repositories {
   mavenLocal()
   maven { url "http://cloudant.github.io/cloudant-sync-eap/repository/" }
   mavenCentral()
}

dependencies {
   compile group: 'com.cloudant', name: 'cloudant-sync-datastore-core', version:'0.13.2'
   compile group: 'com.cloudant', name: 'cloudant-sync-datastore-android', version:'0.13.2'
   compile group: 'com.cloudant', name: 'cloudant-sync-datastore-android-encryption', version:'0.13.2'
}
```

2. Download the SQLCipher for Android v3.2 (https://www.zetetic.net/sqlcipher/open-source/) **.jar** and **.so** binary files and include them in your application in the appropriate folders within your app structure:
   - Add libraries. Add the shared library files and SQLCipher archive to the **jniLibs** folder under your Android app directory.
   - Add the required ICU compressed file to the assets folder in your app.
   - Add **sqlcipher.jar** as a file dependency. From the app folder menu in Android studio, select the **Dependencies** tab under **Open Module Settings**.
3. Initialize your local store for encryption with a key provider.

   > **Warning:** If you change the password after you create the database, an error occurs because the existing database cannot be decrypted. You cannot change your password after the database is encrypted. You must delete the database to change passwords.

   BEFORE (with IMFData/CloudantToolkit):

```
// Get reference to DataManager
DataManager manager = DataManager.getInstance();

// Initalize a key provider
KeyProvider keyProvider = new AndroidKeyProvider(getContext(),"password","identifier");

// Create local store
String databaseName = "automobiledb";
Task<Store> storeTask = manager.localStore(databaseName, keyProvider);
storeTask.continueWith(new Continuation<Store, Void >() {
   @Override
   public Void then(Task<Store> task) throws Exception {
      if (task.isFaulted()) {
         // Handle error
      } else {
         // Do something with Store
         Store store = task.getResult();
      }
      return null;
   }
});
```

   AFTER (with Cloudant Sync):

```
// Load SQLCipher libs
SQLiteDatabase.loadLibs(context);

// Create DatastoreManager
File path = context.getDir("databasedir", Context.MODE_PRIVATE);
DatastoreManager manager = new DatastoreManager(path.getAbsolutePath());

// Create encrypted local store
String name = "automobiledb";

KeyProvider keyProvider = new AndroidKeyProvider(context,"passw0rd","identifier");
Datastore datastore = manager.openDatastore(name, keyProvider);
```

4. When you are replicating data with an encrypted local store, you must pass a KeyProvider object into the `pullReplicationForStore()` or `pushReplicationForStore()` method.

   BEFORE (with IMFData/CloudantToolkit):

```
//Get reference to data manager
DataManager manager = DataManager.getInstance();
String databaseName = "automobiledb";

// Initalize a key provider
KeyProvider keyProvider = new AndroidKeyProvider(getContext(),"password","identifier");

// pull replication
Task<PushReplication> pullTask = manager.pullReplicationForStore(databaseName, keyProvider);

// push replication
Task<PushReplication> pushTask = manager.pushReplicationForStore(databaseName, keyProvider);
```

   AFTER (with Cloudant Sync):

   Replication with an encrypted database requires no changes from replication with an unencrypted database.

Setting user permissions

You can set user permissions on remote databases.

BEFORE (with IMFData/CloudantToolkit):

Objective-C

```
// Get reference to data manager
IMFDataManager *manager = [IMFDataManager sharedInstance];

// Set permissions for current user on a store
[manager setCurrentUserPermissions: DB_ACCESS_GROUP_MEMBERS forStoreName: @"automobiledb" completionHander:^(BOOL suc
cess, NSError *error) {
    if(error){
        // Handle error
    }else{
        // setting permissions was successful
    }
}];
```

Swift

```
// Get reference to data manager
let manager = IMFDataManager.sharedInstance()

// Set permissions for current user on a store
manager.setCurrentUserPermissions(DB_ACCESS_GROUP_MEMBERS, forStoreName: "automobiledb") { (success:Bool, error:NSError!)
-> Void in
    if nil != error {
        // Handle error
    } else {
        // setting permissions was successful
    }
}
```

Java

```
Task<Boolean> permissionsTask = manager.setCurrentUserPermissions(DataManager.DB_ACCESS_GROUP_MEMBERS, "automobiledb"
);

permissionsTask.continueWith(new Continuation<Boolean, Object>() {
   @Override
   public Object then(Task<Boolean> task) throws Exception {
      if(task.isFaulted()){
         // Handle error
      }else{
        // setting permissions was successful
      }
      return null;
   }
});
```

AFTER (with Cloudant Sync):

You cannot set user permissions from the mobile device. You must set permissions with the Cloudant dashboard or server-side code. For a sample of how to integrate MobileFirst OAuth tokens with Cloudant Security, see the Bluelist sample (https://github.ibm.com/MFPSamples/BlueList-On-Premise).

Modeling data

Cloudant stores data as JSON documents. To store data as objects in your application, use the included data object mapper class that maps native objects to the underlying JSON document format.

- iOS: Cloudant stores data as JSON documents. The CloudantToolkit framework provided an object mapper to map between native objects and JSON documents. The CDTDatastore API does not provide this feature. The snippets in the following sections demonstrate how to use CDTDatastore objects to accomplish the same operations.
- Android: AndroidCloudant stores data as JSON documents. The CloudantToolkit API provided an object mapper to map between native objects and JSON documents. Cloudant Sync does not provide this feature. The snippets in the following sections demonstrate how to use DocumentRevision objects to accomplish the same operations.

Performing CRUD operations

You can modify the content of a data store.

- For more details on `create`, `retrieve`, `update`, and `delete` (CRUD) operations, see CDTDatastore CRUD documentation (https://github.com/cloudant/CDTDatastore/blob/master/doc/crud.md).
- For `create`, `retrieve`, `update`, and `delete` (CRUD) operations on a remote store, see the Cloudant Document API (https://docs.cloudant.com/document.html).

Creating data

BEFORE

Objective-C

```
// Use an existing store
CDTStore *store = existingStore;

// Create your Automobile to save
Automobile *automobile = [[Automobile alloc] initWithMake:@"Toyota" model:@"Corolla" year: 2006];

[store save:automobile completionHandler:^(id savedObject, NSError *error) {
   if (error) {
      // save was not successful, handler received an error
   } else {
      // use the result
      Automobile *savedAutomobile = savedObject;
      NSLog(@"saved revision: %@", savedAutomobile);
   }
}];
```

Swift

```swift
// Use an existing store
let store:CDTStore = existingStore

// Create your object to save
let automobile = Automobile(make: "Toyota", model: "Corolla", year: 2006)

store.save(automobile, completionHandler: { (savedObject:AnyObject!, error:NSError!) -> Void in
  if nil != error {
    //Save was not successful, handler received an error
  } else {
    // Use the result
    print("Saved revision: \(savedObject)")
  }
})
```

Java

```java
// Use an existing store
Store store = existingStore;

// Create your object to save
Automobile automobile = new Automobile("Toyota", "Corolla", 2006);

// Save automobile to store
Task<Object> saveTask = store.save(automobile);
saveTask.continueWith(new Continuation<Object, Void>() {
  @Override
  public Void then(Task<Object> task) throws Exception {
    if (task.isFaulted()) {
      // save was not successful, task.getError() contains the error
    } else {
      // use the result
      Automobile savedAutomobile = (Automobile) task.getResult();
    }
    return null;
  }
});
```

AFTER

```objc
// Use an existing store
CDTDatastore *datastore = existingDatastore;

// Create document body
CDTMutableDocumentRevision * revision = [CDTMutableDocumentRevision revision];
revision.body = @{@"@datatype" : @"Automobile", @"make" :@"Toyota", @"model": @"Corolla", @"year" : @2006};

NSError *error = nil;
CDTDocumentRevision *createdRevision = [datastore createDocumentFromRevision:revision error:&error];

if (error) {
  // save was not successful, handler received an error
} else {
  // use the result
  NSLog(@"Revision: %@", createdRevision);
}
```

Swift

```
// Use an existing store
let datastore:CDTDatastore = existingDatastore

// Create document body
let revision = CDTMutableDocumentRevision()
revision.setBody(["make":"Toyota","model":"Corolla","year":2006])

var createdRevision:CDTDocumentRevision?
do{
   createdRevision = try datastore.createDocumentFromRevision(revision)
   NSLog("Revision: \(createdRevision)");
}catch let error as NSError{
   // Handle error
}
```

Java

```
// Use an existing store
Datastore datastore = existingStore;

// Create document body
Map<String, Object> body = new HashMap<String, Object>();
body.put("@datatype", "Automobile");
body.put("make", "Toyota");
body.put("model", "Corolla");
body.put("year", 2006);

// Create revision and set body
MutableDocumentRevision revision  = new MutableDocumentRevision();
revision.body = DocumentBodyFactory.create(body);

// Save revision to store
DocumentRevision savedRevision = datastore.createDocumentFromRevision(revision);
```

Reading data

BEFORE

Objective-C

```
CDTStore *store = existingStore;
NSString *automobileId = existingAutomobileId;

// Fetch Autombile from Store
[store fetchById:automobileId completionHandler:^(id object, NSError *error) {
   if (error) {
      // fetch was not successful, handler received an error
   } else {
      // use the result
      Automobile *savedAutomobile = object;
      NSLog(@"fetched automobile: %@", savedAutomobile);
   }
}];
```

Swift

```
// Using an existing store and Automobile
let store:CDTStore = existingStore
let automobileId:String = existingAutomobileId

// Fetch Autombile from Store
store.fetchById(automobileId, completionHandler: { (object:AnyObject!, error:NSError!) -> Void in
   if nil != error {
      // Fetch was not successful, handler received an error
   } else {
      // Use the result
      let savedAutomobile:Automobile = object as! Automobile
      print("Fetched automobile: \(savedAutomobile)")
   }
})
```

Java

```
// Use an existing store and documentId
Store store = existingStore;
String automobileId = existingAutomobileId;

// Fetch the automobile from the store
Task<Object> fetchTask = store.fetchById(automobileId);
fetchTask.continueWith(new Continuation<Object, Void>() {
   @Override
   public Void then(Task<Object> task) throws Exception {
      if (task.isFaulted()) {
         // fetch was not successful, task.getError() contains the error
      } else {
         // use the result
         Automobile fetchedAutomobile = (Automobile) task.getResult();
      }
      return null;
   }
});
```

AFTER

Objective-C

```
// Use an existing store and documentId
CDTDatastore *datastore = existingDatastore;
NSString *documentId = existingDocumentId;

// Fetch the CDTDocumentRevision from the store
NSError *error = nil;
CDTDocumentRevision *fetchedRevision = [datastore getDocumentWithId:documentId error:&error];

if (error) {
   // fetch was not successful, handler received an error
} else {
   // use the result
   NSLog(@"Revision: %@", fetchedRevision);
}
```

Swift

```
// Use an existing store and documentId
let datastore:CDTDatastore = existingDatastore
let documentId:String = existingDocumentId

var fetchedRevision:CDTDocumentRevision?
do{
   fetchedRevision = try datastore.getDocumentWithId(documentId)
   NSLog("Revision: \(fetchedRevision)");
}catch let error as NSError{
   // Handle error
}
```

Java

```
// Use an existing store and documentId
Datastore datastore = existingStore;
String documentId = existingDocumentId;

// Fetch the revision from the store
DocumentRevision fetchedRevision = datastore.getDocument(documentId);
```

Updating data

BEFORE

Objective-C

```
// Use an existing store and Automobile
CDTStore *store = existingStore;
Automobile *automobile = existingAutomobile;

// Update some of the values in the Automobile
automobile.year = 2015;

// Save Autombile to the store
[store save:automobile completionHandler:^(id savedObject, NSError *error) {
    if (error) {
        // sasve was not successful, handler received an error
    } else {
        // use the result
        Automobile *savedAutomobile = savedObject;
        NSLog(@"saved automobile: %@", savedAutomobile);
    }
}];
```

Swift

```
// Use an existing store and Automobile
let store:CDTStore = existingStore
let automobile:Automobile = existingAutomobile

// Update some of the values in the Automobile
automobile.year = 2015

// Save Autombile to the store
store.save(automobile, completionHandler: { (savedObject:AnyObject!, error:NSError!) -> Void in
    if nil != error {
        // Update was not successful, handler received an error
    } else {
        // Use the result
        let savedAutomobile:Automobile = savedObject as! Automobile
        print("Updated automobile: \(savedAutomobile)")
    }
})
```

Java

```
// Use an existing store and Automobile
Store store = existingStore;
Automobile automobile = existingAutomobile;

// Update some of the values in the Automobile
automobile.setYear(2015);

// Save automobile to store
Task<Object> saveTask = store.save(automobile);
saveTask.continueWith(new Continuation<Object, Void>() {
    @Override
    public Void then(Task<Object> task) throws Exception {
        if (task.isFaulted()) {
            // save was not successful, task.getError() contains the error
        } else {
            // use the result
            Automobile savedAutomobile = (Automobile) task.getResult();
        }
        return null;
    }
});
```

AFTER

Objective-C

```
// Use an existing store and document
CDTDatastore *datastore = existingDatastore;
CDTMutableDocumentRevision *documentRevision = [existingDocumentRevision mutableCopy];

// Update some of the values in the revision
[documentRevision.body setValue:@2015 forKey:@"year"];

NSError *error = nil;
CDTDocumentRevision *updatedRevision = [datastore updateDocumentFromRevision:documentRevision error:&error];
if (error) {
   // save was not successful, handler received an error
} else {
   // use the result
   NSLog(@"Revision: %@", updatedRevision);
}
```

Swift

```
// Use an existing store and document
let datastore:CDTDatastore = existingDatastore
let documentRevision:CDTMutableDocumentRevision = existingDocumentRevision.mutableCopy()

// Update some of the values in the revision
documentRevision.body()["year"] = 2015

var updatedRevision:CDTDocumentRevision?
do{
   updatedRevision = try datastore.updateDocumentFromRevision(documentRevision)
   NSLog("Revision: \(updatedRevision)");
}catch let error as NSError{
   // Handle error
}
```

Java

```
// Use an existing store and documentId
// Use an existing store
Datastore datastore = existingStore;

// Make a MutableDocumentRevision from the existing revision
MutableDocumentRevision revision = existingRevision.mutableCopy();

// Update some of the values in the revision
Map<String, Object> body = revision.getBody().asMap();
body.put("year", 2015);
revision.body = DocumentBodyFactory.create(body);

// Save revision to store
DocumentRevision savedRevision = datastore.updateDocumentFromRevision(revision);
```

Deleting data

To delete an object, pass the object that you want to delete to the store.

BEFORE

Objective-C

```
// Using an existing store and Automobile
CDTStore *store = existingStore;
Automobile *automobile = existingAutomobile;

// Delete the Automobile object from the store
[store delete:automobile completionHandler:^(NSString *deletedObjectId, NSString *deletedRevisionId, NSError *error) {
   if (error) {
      // delete was not successful, handler received an error
   } else {
      // use the result
      NSLog(@"deleted Automobile doc-%@-rev-%@", deletedObjectId, deletedRevisionId);
   }
}];
```

Swift

```
// Using an existing store and Automobile
let store:CDTStore = existingStore
let automobile:Automobile = existingAutomobile

// Delete the Automobile object
store.delete(automobile, completionHandler: { (deletedObjectId:String!, deletedRevisionId:String!, error:NSError!) -> Void in
    if nil != error {
        // delete was not successful, handler received an error
    } else {
        // use the result
        print("deleted document doc-\(deletedObjectId)-rev-\(deletedRevisionId)")
    }
})
```

Java

```
// Use an existing store and automobile
Store store = existingStore;
Automobile automobile = existingAutomobile;

// Delete the automobile from the store
Task<String> deleteTask = store.delete(automobile);
deleteTask.continueWith(new Continuation<String, Void>() {
    @Override
    public Void then(Task<String> task) throws Exception {
        if (task.isFaulted()) {
            // delete was not successful, task.getError() contains the error
        } else {
            // use the result
            String deletedAutomobileId = task.getResult();
        }
        return null;
    }
});
```

AFTER

Objective-C

```
// Use an existing store and revision
CDTDatastore *datastore = existingDatastore;
CDTDocumentRevision *documentRevision = existingDocumentRevision;

// Delete the CDTDocumentRevision from the store
NSError *error = nil;
CDTDocumentRevision *deletedRevision = [datastore deleteDocumentFromRevision:documentRevision error:&error];
if (error) {
    // delete was not successful, handler received an error
} else {
    // use the result
    NSLog(@"deleted document: %@", deletedRevision);
}
```

Swift

```
// Use an existing store and revision
let datastore:CDTDatastore = existingDatastore
let documentRevision:CDTDocumentRevision = existingDocumentRevision

var deletedRevision:CDTDocumentRevision?
do{
    deletedRevision = try datastore.deleteDocumentFromRevision(documentRevision)
    NSLog("Revision: \(deletedRevision)");
}catch let error as NSError{
    // Handle error
}
```

Java

```
// Use an existing store and revision
Datastore datastore = existingStore;
BasicDocumentRevision documentRevision = (BasicDocumentRevision) existingDocumentRevision;

// Delete revision from store
DocumentRevision deletedRevision = datastore.deleteDocumentFromRevision(documentRevision);
```

Creating indexes

To perform queries, you must create an index.

- iOS: For more details, see CDTDatastore Query documentation (https://github.com/cloudant/CDTDatastore/blob/master/doc/query.md). For query operations on a remote store, see the Cloudant Query API (https://docs.cloudant.com/cloudant_query.html).
- Android: For more details, see Cloudant Sync Query documentation (https://github.com/cloudant/sync-android/blob/master/doc/query.md). For CRUD operations on a remote store, see Cloudant's Query API (https://docs.cloudant.com/cloudant_query.html).

1. Create an index that includes the data type. Indexing with the data type is useful when an object mapper is set on the data store.

   BEFORE

   Objective-C

   ```
   // Use an existing data store
   CDTStore *store = existingStore;

   // The data type to use for the Automobile class
   NSString *dataType = [store.mapper dataTypeForClassName:NSStringFromClass([Automobile class])];

   // Create the index
   [store createIndexWithDataType:dataType fields:@[@"year", @"make"] completionHandler:^(NSError *error) {
     if(error){
       // Handle error
     }else{
       // Continue application flow
     }
   }];
   ```

   Swift

   ```
   // A store that has been previously created.
   let store:CDTStore = existingStore

   // The data type to use for the Automobile class
   let dataType:String = store.mapper.dataTypeForClassName(NSStringFromClass(Automobile.classForCoder()))

   // Create the index
   store.createIndexWithDataType(dataType, fields: ["year","make"]) { (error:NSError!) -> Void in
     if nil != error {
       // Handle error
     } else {
       // Continue application flow
     }
   }
   ```

   Java

```
// Use an existing data store
Store store = existingStore;

// The data type to use for the Automobile class
String dataType = store.getMapper().getDataTypeForClassName(Automobile.class.getCanonicalName());

// The fields to index.
List<IndexField> indexFields = new ArrayList<IndexField>();
indexFields.add(new IndexField("year"));
indexFields.add(new IndexField("make"));

// Create the index
Task<Void> indexTask = store.createIndexWithDataType(dataType, indexFields);
indexTask.continueWith(new Continuation<Void, Void>() {
   @Override
   public Void then(Task<Void> task) throws Exception {
      if(task.isFaulted()){
         // Handle error
      }else{
         // Continue application flow
      }
      return null;
   }
});
```

AFTER

Objective-C

```
// A store that has been previously created.
CDTDatastore *datastore = existingDatastore;

NSString *indexName = [datastore ensureIndexed:@[@"@datatype", @"year", @"make"] withName:@"automobileindex"];
if(!indexName){
   // Handle error
}
```

Swift

```
// A store that has been previously created.
let datastore:CDTDatastore = existingDatastore

// Create the index
let indexName:String? = datastore.ensureIndexed(["@datatype","year","make"], withName: "automobileindex")
if(indexName == nil){
   // Handle error
}
```

Java

```
// Use an existing store
Datastore datastore = existingStore;

// Create an IndexManager
IndexManager indexManager = new IndexManager(datastore);

// The fields to index.
List<Object> indexFields = new ArrayList<Object>();
indexFields.add("@datatype");
indexFields.add("year");
indexFields.add("make");

// Create the index
indexManager.ensureIndexed(indexFields, "automobile_index");
```

2. Delete indexes.

BEFORE

Objective-C

```
// Use an existing data store
CDTStore *store = existingStore;
NSString *indexName = existingIndexName;

// Delete the index
[store deleteIndexWithName:indexName completionHandler:^(NSError *error) {
    if(error){
        // Handle error
    }else{
        // Continue application flow
    }
}];
```

Swift

```
// Use an existing store
let store:CDTStore = existingStore

// The data type to use for the Automobile class
let dataType:String = store.mapper.dataTypeForClassName(NSStringFromClass(Automobile.classForCoder()))

// Delete the index
store.deleteIndexWithDataType(dataType, completionHandler: { (error:NSError!) -> Void in
    if nil != error {
        // Handle error
    } else {
        // Continue application flow
    }
})
```

Java

```
// Use an existing data store
Store store = existingStore;
String indexName = existingIndexName;

// Delete the index
Task<Void> indexTask = store.deleteIndex(indexName);
indexTask.continueWith(new Continuation<Void, Void>() {
    @Override
    public Void then(Task<Void> task) throws Exception {
        if(task.isFaulted()){
            // Handle error
        }else{
            // Continue application flow
        }
        return null;
    }
});
```

AFTER

Objective-C

```
// Use an existing store
CDTDatastore *datastore = existingDatastore;
NSString *indexName = existingIndexName;

// Delete the index
BOOL success = [datastore deleteIndexNamed:indexName];
if(!success){
    // Handle error
}
```

Swift

```
// A store that has been previously created.
let datastore:CDTDatastore = existingDatastore
let indexName:String = existingIndexName

// Delete the index
let success:Bool = datastore.deleteIndexNamed(indexName)
if(!success){
    // Handle error
}
```

Java

```
// Use an existing store
Datastore datastore = existingStore;
String indexName = existingIndexName;
IndexManager indexManager = existingIndexManager;

// Delete the index
indexManager.deleteIndexNamed(indexName);
```

Querying data

After you create an index, you can query the data in your database.

- iOS: For more details, see CDTDatastore Query documentation (https://github.com/cloudant/CDTDatastore/blob/master/doc/query.md).
- Android: For more details, see Cloudant Sync Query documentation (https://github.com/cloudant/sync-android/blob/master/doc/query.md).
- For query operations on a remote store, see the Cloudant Query API (https://docs.cloudant.com/cloudant_query.html).

iOS

BEFORE (with IMFData/CloudantToolkit):

Objective-C

```
// Use an existing store
CDTStore *store = existingStore;

NSPredicate *queryPredicate = [NSPredicate predicateWithFormat:@"(year = 2006)"];
CDTCloudantQuery *query = [[CDTCloudantQuery alloc] initDataType:[store.mapper dataTypeForClassName:NSStringFromClass([Automo
bile class])] withPredicate:queryPredicate];

[store performQuery:query completionHandler:^(NSArray *results, NSError *error) {
    if(error){
        // Handle error
    }else{
        // Use result of query.  Result will be Automobile objects.
    }
}];
```

Swift

```
// Use an existing store
let store:CDTStore = existingStore

let queryPredicate:NSPredicate = NSPredicate(format:"(year = 2006)")
let query:CDTCloudantQuery = CDTCloudantQuery(dataType: "Automobile", withPredicate: queryPredicate)

store.performQuery(query, completionHandler: { (results:[AnyObject]!, error:NSError!) -> Void in
    if nil != error {
        // Handle error
    } else {
        // Use result of query.  Result will be Automobile objects.
    }
})
```

AFTER (with Cloudant Sync):

Objective-C

```
// Use an existing store
CDTDatastore *datastore = existingDatastore;

CDTQResultSet *results = [datastore find:@{@"@datatype" : @"Automobile", @"year" : @2006}];
if(results){
    // Use results
}
```

```
// Use an existing store
let datastore:CDTDatastore = existingDatastore

let results:CDTQResultSet? = datastore.find(["@datatype" : "Automobile", "year" : 2006])
if(results == nil){
    // Handle error
}
```

Android

To run a query for objects, create a Cloudant query with the query filters on data type. Run the query against a Store object.

BEFORE (with IMFData/CloudantToolkit):

```
// Use an existing store
Store store = existingStore;

// Create data type predicate
Map<String, Object> dataTypeEqualityOpMap = new HashMap<String, Object>();
dataTypeEqualityOpMap.put("$eq", "Automobile");

Map<String, Object> dataTypeSelectorMap = new HashMap<String, Object>();
dataTypeSelectorMap.put("@datatype", dataTypeEqualityOpMap);

// Create year predicate
Map<String, Object> yearEqualityOpMap = new HashMap<String, Object>();
yearEqualityOpMap.put("$eq", 2006);

Map<String, Object> yearSelectorMap = new HashMap<String, Object>();
yearSelectorMap.put("year", yearEqualityOpMap);

// Add predicates to AND compound predicate
List<Map<String, Object>> andPredicates = new ArrayList<Map<String, Object>>();
andPredicates.add(dataTypeSelectorMap);
andPredicates.add(yearSelectorMap);

Map<String, Object> andOpMap = new HashMap<String, Object>();
andOpMap.put("$and", andPredicates);

Map<String, Object> cloudantQueryMap = new HashMap<String, Object>();
cloudantQueryMap.put("selector", andOpMap);

// Create a Cloudant Query Object
CloudantQuery query = new CloudantQuery(cloudantQueryMap);

// Run the Cloudant Query against a Store
Task<List> queryTask = store.performQuery(query);
queryTask.continueWith(new Continuation<List, Object>() {
    @Override
    public Object then(Task<List> task) throws Exception {
        if(task.isFaulted()){
            // Handle Error
        }else{
            List queryResult = task.getResult();
            // Use queryResult to do something
        }
        return null;
    }
});
```

AFTER (with Cloudant Sync):

```
// Use an existing store
Datastore datastore = existingStore;
IndexManager indexManager = existingIndexManager;

// Create data type predicate
Map<String, Object> dataTypeEqualityOpMap = new HashMap<String, Object>();
dataTypeEqualityOpMap.put("$eq", "Automobile");

Map<String, Object> dataTypeSelectorMap = new HashMap<String, Object>();
dataTypeSelectorMap.put("@datatype", dataTypeEqualityOpMap);

// Create year predicate
Map<String, Object> yearEqualityOpMap = new HashMap<String, Object>();
yearEqualityOpMap.put("$eq", 2006);

Map<String, Object> yearSelectorMap = new HashMap<String, Object>();
yearSelectorMap.put("year", yearEqualityOpMap);

// Add predicates to AND compound predicate
List<Map<String, Object>> andPredicates = new ArrayList<Map<String, Object>>();
andPredicates.add(dataTypeSelectorMap);
andPredicates.add(yearSelectorMap);

Map<String, Object> selectorMap = new HashMap<String, Object>();
selectorMap.put("$and", andPredicates);

// Run the query against a Store
QueryResult result = indexManager.find(selectorMap);
```

Supporting offline storage and synchronization

You can synchronize the data on a mobile device with a remote database instance. You can either pull updates from a remote database to the local database on the mobile device, or push local database updates to a remote database.

- iOS: For more details, see CDTDatastore Replication documentation (https://github.com/cloudant/CDTDatastore/blob/master/doc/replication.md).
- Android For more details, see Cloudant Sync Replication documentation (https://github.com/cloudant/sync-android/blob/master/doc/replication.md). For CRUD operations on a remote store, see the Cloudant Replication API (https://docs.cloudant.com/replication.html).

Running pull replication

BEFORE

Objective-C

```
// store is an existing CDTStore object created using IMFDataManager remoteStore
__block 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
}

// (optionally) monitor replication via polling
while (replicator.isActive) {
    [NSThread sleepForTimeInterval:1.0f];
    NSLog(@"replicator state : %@", [CDTReplicator stringForReplicatorState:replicator.state]);
}
```

Swift

```
// Use an existing store
let store:CDTStore = existingStore

do {
    // store is an existing CDTStore object created using IMFDataManager remoteStore
    let pull:CDTPullReplication = manager.pullReplicationForStore(store.name)
    let replicator:CDTReplicator = try manager.replicatorFactory.oneWay(pull)

    // start replication
    try replicator.start()

    // (optionally) monitor replication via polling
    while replicator.isActive() {
        NSThread.sleepForTimeInterval(1.0)
        print("replicator state : \(CDTReplicator.stringForReplicatorState(replicator.state))")
    }

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

Java

```
// Use an existing store
Store store = existingStore;

// create a pull replication task
// name is the database name of the store being replicated
Task<PullReplication> pullTask = manager.pullReplicationForStore(store.getName());
pullTask.continueWith(new Continuation<PullReplication, Object>() {
    @Override
    public Object then(Task<PullReplication> task) throws Exception {
        if(task.isFaulted()){
            // Handle error
        }else{
            // Start the replication
            PullReplication pull = task.getResult();
            Replicator replicator = ReplicatorFactory.oneway(pull);
            replicator.start();
        }
        return null;
    }
});
```

AFTER

Objective-C

```
// Use an existing datastore
NSURL *remoteStoreUrl = existingRemoteStoreUrl;
CDTDatastoreManager *datastoreManager = existingDatastoreManager;
CDTDatastore *datastore = existingDatastore;

// Create pull replication objects
__block NSError *replicationError;
CDTReplicatorFactory *replicatorFactory = [[CDTReplicatorFactory alloc]initWithDatastoreManager:datastoreManager];
CDTPullReplication *pull = [CDTPullReplication replicationWithSource:remoteStoreUrl target:datastore];
CDTReplicator *replicator = [replicatorFactory oneWay:pull error:&error];
if(replicationError){
    // Handle error
}else{
    // replicator creation was successful
}

[replicator startWithError:&replicationError];
if(replicationError){
    // Handle error
}else{
    // replicator start was successful
}

// (optionally) monitor replication via polling
while (replicator.isActive) {
    [NSThread sleepForTimeInterval:1.0f];
    NSLog(@"replicator state : %@", [CDTReplicator stringForReplicatorState:replicator.state]);
}
```

Swift

```
let remoteStoreUrl:NSURL = existingRemoteStoreUrl
let datastoreManager:CDTDatastoreManager = existingDatastoreManager
let datastore:CDTDatastore = existingDatastore


do {
    // store is an existing CDTStore object created using IMFDataManager remoteStore
    let replicatorFactory = CDTReplicatorFactory(datastoreManager: datastoreManager)
    let pull:CDTPullReplication = CDTPullReplication(source: remoteStoreUrl, target: datastore)
    let replicator:CDTReplicator = try replicatorFactory.oneWay(pull)

    // start replication
    try replicator.start()

    // (optionally) monitor replication via polling
    while replicator.isActive() {
        NSThread.sleepForTimeInterval(1.0)
        print("replicator state : \(CDTReplicator.stringForReplicatorState(replicator.state))")
    }

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

Java

```
// Use an opened Datastore to replicate to
Datastore datastore = existingDatastore;
URI uri = existingURI;

// Create a replicator that replicates changes from the remote
final Replicator replicator = ReplicatorBuilder.pull().from(uri).to(datastore).build();

// Register event listener
replicator.getEventBus().register(new Object() {

    @Subscribe
    public void complete(ReplicationCompleted event) {

        // Handle ReplicationCompleted event
    }

    @Subscribe
    public void error(ReplicationErrored event) {

        // Handle ReplicationErrored event
    }
});

// Start replication
replicator.start();
```

Running push replication

BEFORE

Objective-C

```
/ store is an existing CDTStore object created using IMFDataManager localStore
__block 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
}

// (optionally) monitor replication via polling
while (replicator.isActive) {
    [NSThread sleepForTimeInterval:1.0f];
    NSLog(@"replicator state : %@", [CDTReplicator stringForReplicatorState:replicator.state]);
}
```

Swift

```
// Use an existing store
let store:CDTStore = existingStore

do {
   // store is an existing CDTStore object created using IMFDataManager localStore
   let push:CDTPushReplication = manager.pushReplicationForStore(store.name)
   let replicator:CDTReplicator = try manager.replicatorFactory.oneWay(push)

   // Start replication
   try replicator.start()

   // (optionally) monitor replication via polling
   while replicator.isActive() {
      NSThread.sleepForTimeInterval(1.0)
      print("replicator state : \(CDTReplicator.stringForReplicatorState(replicator.state))")
   }
} catch let error as NSError {
   // Handle error
}
```

Java

```
// Use an existing store
Store store = existingStore;

// create a push replication task
// name is the database name of the store being replicated
Task<PushReplication> pushTask = manager.pushReplicationForStore(store.getName());
pushTask.continueWith(new Continuation<PushReplication, Object>() {
   @Override
   public Object then(Task<PushReplication> task) throws Exception {
      if(task.isFaulted()){
         // Handle error
      }else{
         // Start the replication
         PushReplication push = task.getResult();
         Replicator replicator = ReplicatorFactory.oneway(push);
         replicator.start();
      }
      return null;
   }
});
```

AFTER

Objective-C

```
// Use an existing datastore
NSURL *remoteStoreUrl = existingRemoteStoreUrl;
CDTDatastoreManager *datastoreManager = existingDatastoreManager;
CDTDatastore *datastore = existingDatastore;

// Create push replication objects
__block NSError *replicationError;
CDTReplicatorFactory *replicatorFactory = [[CDTReplicatorFactory alloc]initWithDatastoreManager:datastoreManager];
CDTPushReplication *push = [CDTPushReplication replicationWithSource:datastore target:remoteStoreUrl];
CDTReplicator *replicator = [replicatorFactory oneWay:push error:&error];
if(replicationError){
    // Handle error
}else{
    // replicator creation was successful
}

[replicator startWithError:&replicationError];
if(replicationError){
    // Handle error
}else{
    // replicator start was successful
}

// (optionally) monitor replication via polling
while (replicator.isActive) {
    [NSThread sleepForTimeInterval:1.0f];
    NSLog(@"replicator state : %@", [CDTReplicator stringForReplicatorState:replicator.state]);
}
```

Swift

```
let remoteStoreUrl:NSURL = existingRemoteStoreUrl
let datastoreManager:CDTDatastoreManager = existingDatastoreManager
let datastore:CDTDatastore = existingDatastore


do {
    // store is an existing CDTStore object created using IMFDataManager remoteStore
    let replicatorFactory = CDTReplicatorFactory(datastoreManager: datastoreManager)
    let push:CDTPushReplication = CDTPushReplication(source: datastore, target: remoteStoreUrl)
    let replicator:CDTReplicator = try replicatorFactory.oneWay(push)

    // start replication
    try replicator.start()

    // (optionally) monitor replication via polling
    while replicator.isActive() {
        NSThread.sleepForTimeInterval(1.0)
        print("replicator state : \(CDTReplicator.stringForReplicatorState(replicator.state))")
    }

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

Java

```java
// Use an opened Datastore to replicate from
Datastore datastore = existingStore;
URI uri = existingURI;

// Create a replicator that replicates changes from the local
// database to the remote datastore.
final Replicator replicator = ReplicatorBuilder.push().from(datastore).to(uri).build();

// Register event listener
replicator.getEventBus().register(new Object() {

    @Subscribe
    public void complete(ReplicationCompleted event) {

        // Handle ReplicationCompleted event
    }

    @Subscribe
    public void error(ReplicationErrored event) {

        // Handle ReplicationErrored event
    }
});

// Start replication
replicator.start();
```

Last modified on