JSONStore in Cordova applications

Overview

IBM MobileFirst Platform 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 Platform Foundation user documentation website.

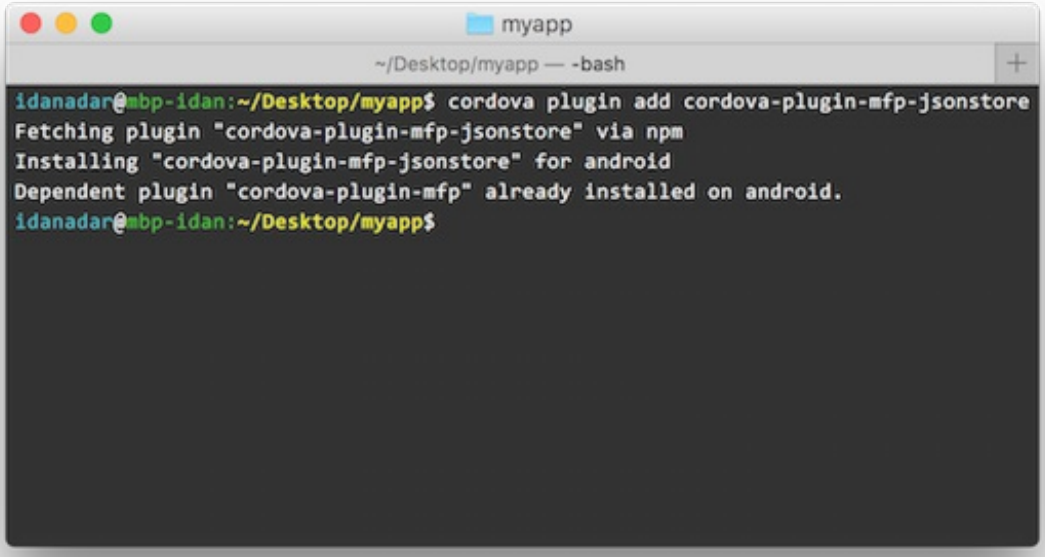
Prerequisite: Make sure the MobileFirst Native SDK was added to the Xcode project. Follow the Adding the MobileFirst Platform Foundation SDK to iOS applications (../../adding-the-mfpf-sdk/cordova/) tutorial.

Jump to:

Adding JSONStore

To add JSONStore plug-in to your Cordova application:

1. Open a **Command-line** window and navigate to your Cordova project folder.
2. Run the command: `cordova plugin add cordova-plugin-mfp-jsonstore`.



Basic Usage

Initialize

Use `init` to start 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.

```
var collections = {
    people : {
        searchFields: {name: 'string', age: 'integer'}
    }
};

WL.JSONStore.init(collections).then(function (collections) {
    // handle success - collection.people (people's collection)
}).fail(function (error) {
    // handle failure
});
```

> 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.

Get

Use `get` to create an accessor to the collection. You must call `init` before you call get otherwise the result of `get` will be undefined.

```
var collectionName = 'people';
var people = WL.JSONStore.get(collectionName);
```

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

Add

Use `add` to store data as documents inside a collection

```
var collectionName = 'people';
var options = {};
var data = {name: 'yoel', age: 23};

WL.JSONStore.get(collectionName).add(data, options).then(function () {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

Find

- Use `find` to locate a document inside a collection by using a query.
- Use `findAll` to retrieve all the documents inside a collection.
- Use `findById` to search by the document unique identifier.

The default behavior for find is to do a "fuzzy" search.

```
var query = {name: 'yoel'};
var collectionName = 'people';
var options = {
  exact: false, //default
  limit: 10 // returns a maximum of 10 documents, default: return every document
};

WL.JSONStore.get(collectionName).find(query, options).then(function (results) {
    // handle success - results (array of documents found)
}).fail(function (error) {
    // handle failure
});
```

Replace

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

```
var document = {
  _id: 1, json: {name: 'chevy', age: 23}
};
var collectionName = 'people';
var options = {};

WL.JSONStore.get(collectionName).replace(document, options).then(function (numberOfDocsReplaced) {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

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

Remove

Use `remove` to delete a document from a collection.
Documents are not erased from the collection until you call push.

> For more information, see the **MobileFirst Adapter Integration** section later in this tutorial

```
var query = {_id: 1};
var collectionName = 'people';
var options = {exact: true};
WL.JSONStore.get(collectionName).remove(query, options).then(function (numberOfDocsRemoved) {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

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.

```
var collectionName = 'people';
WL.JSONStore.get(collectionName).removeCollection().then(function (removeCollectionReturnCode) {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

Advanced Usage

Destroy

Use `destroy` 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)

```
var collectionName = 'people';
WL.JSONStore.destroy().then(function () {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

Security

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

Data encryption is only available on Android, iOS, Windows 8.1 Universal and Windows 10 UWP environments.
Some security metadata is stored in the *keychain* (iOS), *shared preferences* (Android) or the *credential locker* (Windows 8.1).
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 `closeAll` to lock access to all the collections until you call `init` again. If you think of `init` as a login function you can think of `closeAll` as the corresponding logout function. Use `changePassword` to change the password.

```
var collections = {
  people: {
    searchFields: {name: 'string'}
  }
};
var options = {password: '123'};
WL.JSONStore.init(collections, options).then(function () {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

Multiple User Support

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

```
var collections = {
  people: {
    searchFields: {name: 'string'}
  }
};
var options = {username: 'yoel'};
WL.JSONStore.init(collections, options).then(function () {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

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 `WLResourceRequest` or `jQuery.ajax` 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;
}
```

Initialize a collection linked to a MobileFirst adapter

```
var collections = {
  people : {
    searchFields : {name: 'string', age: 'integer'},
    adapter : {
      name: 'People',
      add: 'addPerson',
      remove: 'removePerson',
      replace: 'replacePerson',
      load: {
        procedure: 'getPeople',
        params: [],
        key: 'peopleList'
      }
    }
  }
}

var options = {};
WL.JSONStore.init(collections, options).then(function () {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

Load data from MobileFirst Adapter

When `load` is called, JSONStore uses some metadata about the adapter (**name** and **procedure**), which you previously passed to `init`, to determine what data to get from the adapter and eventually store it.

```
var collectionName = 'people';
WL.JSONStore.get(collectionName).load().then(function (loadedDocuments) {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

Get Push Required (Dirty Documents)

Calling `getPushRequired` returns an array of so called *"dirty documents"*, which are documents that have local modifications that do not exist on the back-end system. These documents are sent to the MobileFirst adapter when `push` is called.

```
var collectionName = 'people';
WL.JSONStore.get(collectionName).getPushRequired().then(function (dirtyDocuments) {
    // handle success
}).fail(function (error) {
    // handle failure
});
```

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

Push

`push` sends the documents that changed to the correct MobileFirst adapter procedure (i.e., `addPerson` is called with a document that was added locally). This mechanism is based on the last operation that is associated with the document that changed and the adapter metadata that is passed to `init`.

```
var collectionName = 'people';
WL.JSONStore.get(collectionName).push().then(function (response) {
    // handle success
    // response is an empty array if all documents reached the server
    // response is an array of error responses if some documents failed to reach the server
}).fail(function (error) {
    // handle failure
});
```

Enhance

Use `enhance` to extend the core API to fit your needs, by adding functions to a collection prototype. This example shows how to use `enhance` to add the function `getValue` that works on the `keyvalue` collection. It takes a `key` (string) as it's only parameter and returns a single result.

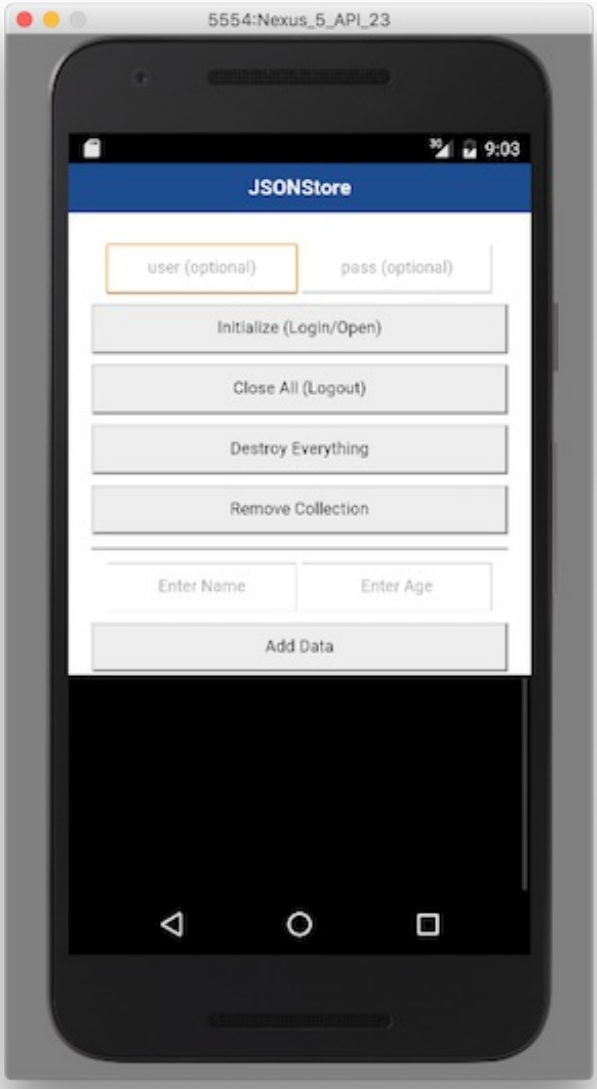
```
var collectionName = 'keyvalue';
WL.JSONStore.get(collectionName).enhance('getValue', function (key) {
    var deferred = $.Deferred();
    var collection = this;
    //Do an exact search for the key
    collection.find({key: key}, {exact:true, limit: 1}).then(deferred.resolve, deferred.reject);
    return deferred.promise();
});

//Usage:
var key = 'myKey';
WL.JSONStore.get(collectionName).getValue(key).then(function (result) {
    // handle success
    // result contains an array of documents with the results from the find
}).fail(function () {
    // handle failure
});
```

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

Sample application

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

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



Sample usage

1. Use either Maven or MobileFirst Developer CLI to build and deploy the adapter (../../creating-adapters/).
2. From the command-line, navigate to the project's root folder.
3. Add a platform by running the `cordova platform add` command.
4. Run the Cordova application by running the `cordova run` command.