Using JSONStore in Cordova applications

Overview

This tutorial is a continuation of the JSONStore Overview tutorial.
Before continuing make sure that jsonstore.jar file is under the libs folder.

The tutorial covers the following topics:

- Basic Usage
  - Initalize
  - Get
  - Add
  - Find
  - Replace
  - Remove
  - Remove Collection
  - Destroy
- Advanced Usage
  - Security
  - Multiple User Support
  - MobileFirst Adapter Integration
  - Enhance
- Sample application

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. 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 [code lang="javascript"] 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 }); [/code]

id="get">Get

Use `get` to create an accessor to the collection. You must call `init` before you call get otherwise the result of get is undefined [code language="javascript"] var collectionName = 'people'; var people = WL.JSONStore.get(collectionName); [/code] The variable `people` can now be used to perform operations on the `people` collection such as `add`, `find`, and `replace`

id="add">Add

Use `add` to store data as documents inside a collection [code language="javascript"] 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 }); [/code]

id="find">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 [code language="javascript"] 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 }); [/code]

id="replace">Replace

Use `replace` to modify documents inside a collection. The field that you use to perform the replacement is `_id`, the document unique identifier. [code language="javascript"] 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 }); [/code] This examples assumes that the document* `{id: 1, json: {name: 'yoel', age: 23} }` is in the collection

id="remove">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 [code language="javascript"] 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 }); [/code]

id="remove-collection">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 [code language="javascript"] var collectionName = 'people'; WL.JSONStore.get(collectionName).removeCollection().then(function (removeCollectionReturnCode) { // handle success }).fail(function (error) { // handle failure }); [/code]

id="destroy">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)

[code language="javascript"] var collectionName = 'people'; WL.JSONStore.destroy().then(function () { // handle success }).fail(function (error) { // handle failure }); [/code]

id="security">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 Phone 8, and Windows 8 environments. Some security metadata are stored in the keychain (iOS), shared preferences (Android), isolated storage (Windows 8 Phone), or the credential locker (Windows 8). 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. [code language="javascript"] var collections = { people: { searchFields: {name: 'string'} } }; var options = {password: '123'}; WL.JSONStore.init(collections, options).then(function () { // handle success }).fail(function (error) { // handle failure }); [/code]

id="multi-user">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** [code language="javascript"] var collections = { people: { searchFields: {name: 'string'} } }; var options = {username: 'yoel'}; WL.JSONStore.init(collections, options).then(function () { // handle success }).fail(function (error) { // handle failure }); [/code]

id="adapter">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 `WL.Client.invokeProcedure` 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`. [code language="javascript"] 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; } [/code]

>Initialize a collection linked to a MobileFirst adapter

[code language="javascript"] 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 }); [/code]

>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. [code language="javascript"] var collectionName = 'people'; WL.JSONStore.get(collectionName).load().then(function (loadedDocuments) { // handle success }).fail(function (error) { // handle failure }); [/code]

> Get Push Required (Dirty Documents)

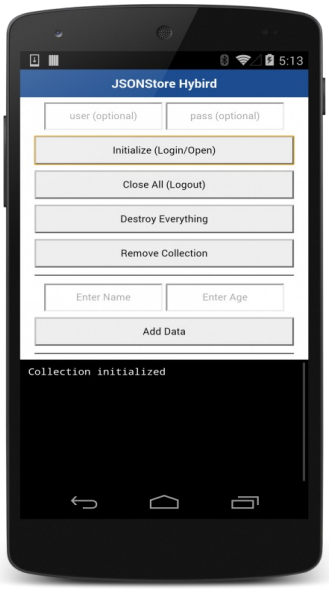
Calling `getPushRequired` returns and array of so called "dirty documents", which are documents that have local modifications that do not exist on the back-end system. These documents are sent to the MobileFirst adapter when `push` is called. [code language="javascript"] var collectionName = 'people'; WL.JSONStore.get(collectionName).getPushRequired().then(function (dirtyDocuments) { // handle success }).fail(function (error) { // handle failure }); [/code] 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`. [code language="javascript"] 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 }); [/code]

id="enhance">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 adde the function `getValue` that works on the `keyvalue` collection. It takes a `key` (string) as it's only parameter and returns a single result. [code language="javascript"] 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 }); [/code]



id="sample">Sample application

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/JSONStore) the MobileFirst project. The MobileFirst project contains an application that demonstrates the use of JSONStore in a hybrid environment.

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