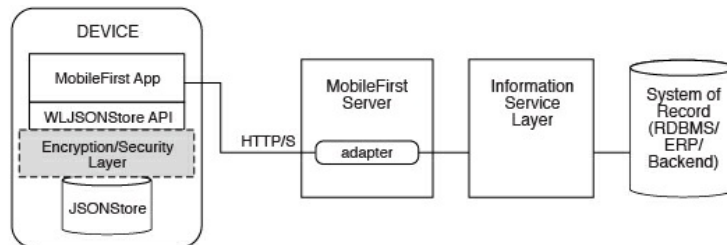# JSONStore

Overview

IBM MobileFirst 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.

Because it is familiar to developers, relational database terminology is used in this documentation at times to help explain JSONStore. There are many differences between a relational database and JSONStore however. For example, the strict schema that is used to store data in relational databases is different from JSONStore's approach. With JSONStore, you can store any JSON content, and index the content that you need to search.

Key features

- Data indexing for efficient searching
- Mechanism for tracking local-only changes to the stored data
- Support for multiple users
- AES 256 encryption of stored data provides security and confidentiality. You can segment protection by user with password-protection, in the case of more than one user on a single device.

A single store can have many collections, and each collection can have many documents. It is also possible to have a MobileFirst application that contains multiple stores. For information, see JSONStore multiple user support.

Support level

- JSONStore is supported in Native iOS and Android applications (no support for native Windows (Universal and UWP)).
- JSONStore is supported in Cordova iOS, Android and Windows (Universal and UWP) applications.

Jump to

- General JSONStore terminology
- Features table
- Multiple User Support
- Security
- Performance
- Concurrency
- Analytics
- Working with External Data
- Troubleshooting
- API Usage

General JSONStore Terminology

Document

A document is the basic building block of JSONStore.

A JSONStore document is a JSON object with an automatically generated identifier (`_id`) and JSON data. It is similar to a record or a row in database terminology. The value of `_id` is always a unique integer inside a specific collection. Some functions like `add`, `replace`, and `remove` in the `JSONStoreInstance` class take an Array of Documents/Objects. These methods are useful to perform operations on various Documents/Objects at a time.

Single document

```
var doc = { _id: 1, json: {name: 'carlos', age: 99} };
```

Array of documents

```
var docs = [
  { _id: 1, json: {name: 'carlos', age: 99} },
  { _id: 2, json: {name: 'tim', age: 100} }
]
```

Collection

A JSONStore collection is similar to a table, in database terminology.

The below code example is not the way that the documents are stored on disk, but it is a good way to visualize what a collection looks like at a high level.

```
[
    { _id: 1, json: {name: 'carlos', age: 99} },
    { _id: 2, json: {name: 'tim', age: 100} }
]
```

Store

A store is the persistent JSONStore file that contains one or more collections.

A store is similar to a relational database, in database terminology. A store is also referred to as a JSONStore.

Search fields

A search field is a key/value pair.

Search fields are keys that are indexed for fast lookup times, similar to column fields or attributes, in database terminology.

Extra search fields are keys that are indexed but that are not part of the JSON data that is stored. These fields define the key whose values (in the JSON collection) are indexed and can be used to search more quickly.

Valid data types are: string, boolean, number, and integer. These types are only type hints, there is no type validation. Furthermore, these types determine how indexable fields are stored. For example, `{age: 'number'}` will index 1 as 1.0 and `{age: 'integer'}` will index 1 as 1.

Search fields and extra search fields

```
var searchField = {name: 'string', age: 'integer'};
var additionalSearchField = {key: 'string'};
```

It is only possible to index keys inside an object, not the object itself. Arrays are handled in a pass-through fashion, meaning that you cannot index an array or a specific index of the array (arr[n]), but you can index objects inside an array.

Indexing values inside an array

```
var searchFields = {
    'people.name' : 'string', // matches carlos and tim on myObject
    'people.age' : 'integer' // matches 99 and 100 on myObject
};

var myObject = {
    people : [
        {name: 'carlos', age: 99},
        {name: 'tim', age: 100}
    ]
};
```

Queries

Queries are objects that use search fields or extra search fields to look for documents.

These examples presumes that the name search field is of type string and the age search field is of type integer.

Find documents with `name` that matches `carlos`

```
var query1 = {name: 'carlos'};
```

Find documents with `name` that matches `carlos` and `age` matches `99`

```
var query2 = {name: 'carlos', age: 99};
```

Query parts

Query parts are used to build more advanced searches. Some JSONStore operations, such as some versions of `find` or `count` take query parts. Everything within a query part is joined by `AND` statements, while query parts themselves are joined by `OR` statements. The search criteria returns a match only if everything within a query part is **true**. You can use more than one query part to search for matches that satisfy one or more of the query parts.

Find with query parts operate only on top-level search fields. For example: `name`, and not `name.first`. Use multiple collections where all search fields are top-level to get around this. The query parts operations that work with non top-level search fields are: `equal`, `notEqual`, `like`, `notLike`, `rightLike`, `notRightLike`, `leftLike`, and `notLeftLike`. The behavior is undetermined if you use non-top-level search fields.

Features table

Compare JSONStore features to those features of other data storage technologies and formats.

JSONStore is a JavaScript API for storing data inside Cordova applications that use the MobileFirst plug-in, an Objective-C API for native iOS applications, and a Java API for native Android applications. For reference, here is a comparison of different JavaScript storage technologies to see how JSONStore compares to them.

JSONStore is similar to technologies such as LocalStorage, Indexed DB, Cordova Storage API, and Cordova File API. The table shows how some features that are provided by JSONStore compare with other technologies. The JSONStore feature is only available on iOS and Android devices and simulators.

Feature	JSONStore	LocalStorage	IndexedDB	Cordova storage API	Cordova file API
Android Support (Cordova & Native Applications)	✔	✔	✔	✔	✔
iOS Support (Cordova & Native Applications)	✔	✔	✔	✔	✔
Windows 8.1 Universal anND Windows 10 UWP	✔	✔	✔	-	✔
Data encryption	✔	-	-	-	-
Maximum Storage	Available Space	~5MB	~5MB	Available Space	Available Space
Reliable Storage (See note)	✔	-	-	✔	✔
Keep Track of Local Changes	✔	-	-	-	-
Multi-user support	✔	-	-	-	-
Indexing	✔	-	✔	✔	-
Type of Storage	JSON documents	Key/value pairs	JSON documents	Relational (SQL)	Strings

Note: Reliable Storage means that your data is not deleted unless one of the following events occurs:

- The application is removed from the device.
- One of the methods that removes data is called.

Multiple User Support

With JSONStore, you can create multiple stores that contain different collections in a single MobileFirst application.

The init (JavaScript) or open (Native iOS and Native Android) API can take an options object with a user name. Different stores are separate files in the file system. The user name is used as the file name of the store. These separate stores can be encrypted with different passwords for security and privacy reasons. Calling the closeAll API removes access to all the collections. It is also possible to change the password of an encrypted store by calling the changePassword API.

An example use case would be various employees that share a physical device (for example an iPad or Android tablet) and MobileFirst application. In addition, if the employees work different shifts and handle private data from different customers while they use the MobileFirst application, multiple user support is useful.

Security

You can secure all of the collections in a store by encrypting them.

To encrypt all of the collections in a store, pass a password to the `init` (JavaScript) or `open` (Native iOS and Native Android) API. If no password is passed, none of the documents in the store collections are encrypted.

Some security artifacts (for example salt) are stored in the keychain (iOS), shared preferences (Android) and the credential locker (Windows Universal 8.1 and Windows 10 UWP). 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). You can choose to encrypt data collections for an application, but you cannot switch between encrypted and plain-text formats, or to mix formats within a store.

The key that protects the data in the store is based on the user password that you provide. The key does not expire, but you can change it by calling the changePassword API.

The data protection key (DPK) is the key that is used to decrypt the contents of the store. The DPK is kept in the iOS keychain even if the application is uninstalled. To remove both the key in the keychain and everything else that JSONStore puts in the application, use the destroy API. This process is not applicable to Android because the encrypted DPK is stored in shared preferences and wiped out when the application is uninstalled.

The first time that JSONStore opens a collection with a password, which means that the developer wants to encrypt data inside the store, JSONStore needs a random token. That random token can be obtained from the client or from the server.

When the localKeyGen key is present in the JavaScript implementation of the JSONStore API, and it has a value of true, a cryptographically secure token is generated locally. Otherwise, the token is generated by contacting the server, thus requiring connectivity to the MobileFirst Server. This token is required only the first time that a store is opened with a password. The native implementations (Objective-C and Java) generate a cryptographically secure token locally by default, or you can pass one through the secureRandom option.

The trade-off is between opening a store offline and trusting the client to generate that random token (less secure), or opening the store with access to the MobileFirst Server (requires connectivity) and trusting the server (more secure).

Security Utilities

The MobileFirst client-side API provides some security utilities to help protect your user's data. Features like JSONStore are great if you want to protect JSON objects. However, it is not recommended to store binary blobs in a JSONStore collection.

Instead, store binary data on the file system, and store the file paths and other metadata inside a JSONStore collection. If you want to protect files like images, you can encode them as base64 strings, encrypt it, and write the output to disk. When it is time to decrypt the data, you can look up the metadata in a JSONStore collection, read the encrypted data from the disk, and decrypt it using the metadata that was stored. This metadata can include the key, salt, Initialization Vector (IV), type of file, path to the file, and others.

> Learn more about JSONStore Security Utilities (security-utilities).

Windows 8.1 Universal and Windows 10 UWP encryption

You can secure all of the collections in a store by encrypting them.

JSONStore uses SQLCipher (http://sqlcipher.net/) as its underlying database technology. SQLCipher is a build of SQLite that is produced by Zetetic, LLC adds a layer of encryption to the database.

JSONStore uses SQLCipher on all platforms. On Android and iOS a free, open source version of SQLCipher is available, known as the Community Edition and is incorporated into the versions of JSONStore that is included in IBM MobileFirst Foundation. The Windows versions of SQLCipher are only available under a commercial license and cannot be directly redistributed by IBM MobileFirst Foundation.

Instead, JSONStore for Windows 8 Universal include SQLite as the underlying database. If you need to encrypt data for either of these platforms, you need to acquire your own version of SQLCipher and swap out the SQLite version that is included in IBM MobileFirst Foundation.

If you do not need encryption, the JSONStore is fully functional (minus encryption) by using the SQLite version in IBM MobileFirst Foundation.

Replacing SQLite with SQLCipher for Windows Universal and Windows UWP

1. Run the SQLCipher for Windows Runtime 8.1/10 extension that comes with the SQLCipher for Windows Runtime Commercial Edition.
2. After the extension finishes installing, locate the SQLCipher version of the **sqlite3.dll** file that was just created. There is one for x86, one for x64, and one for ARM.

   C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1\ExtensionSDKs\SQLCipher.WinRT81\3.0.1\Redist\Retail\<platform>

3. Copy and replace this file to your MobileFirst application.

   <Worklight project name>\apps\<application name>\windows8\native\buildtarget\<platform>

Performance

The following are factors that can affect JSONStore performance.

Network

- Check network connectivity before you perform operations, such as sending all dirty documents to a MobileFirst adapter.
- The amount of data that is sent over the network to a client heavily affects performance. Send only the data that is required by the application, instead of copying everything inside your backend database.
- If you are using a MobileFirst adapter, consider setting the compressResponse flag to true. That way, responses are compressed, which generally uses less bandwidth and has a faster transfer time than without compression.

Memory

- When you use the JavaScript API, JSONStore documents are serialized and deserialized as Strings between the Native (Objective-C, Java, or C#) Layer and the JavaScript Layer. One way to mitigate possible memory issues is by using limit and offset when you use the find API. That way, you limit the amount of memory that is allocated for the results and can implement things like pagination (show X number of results per page).
- Instead of using long key names that are eventually serialized and deserialized as Strings, consider mapping those long key names into smaller ones (for example: `myVeryVeryVerLongKeyName` to `k` or `key`). Ideally, you map them to short key names when you send them from the adapter to the client, and map them to the original long key names when you send data back to the backend.
- Consider splitting the data inside a store into various collections. Have small documents over various collections instead of monolithic documents in a single collection. This consideration depends on how closely related the data is and the use cases for said data.
- When you use the add API with an array of objects, it is possible to run into memory issues. To mitigate this issue, call these methods with fewer JSON objects at a time.
- JavaScript and Java have garbage collectors, while Objective-C has Automatic Reference Counting. Allow it to work, but do not depend on it entirely. Try to null references that are no longer used and use profiling tools to check that memory usage is going down when you expect it to go down.

CPU

- The amount of search fields and extra search fields that are used affect performance when you call the add method, which does the indexing. Only index the values that are used in queries for the find method.
- By default, JSONStore tracks local changes to its documents. This behavior can be disabled, thus saving a few cycles, by setting the `markDirty` flag to **false** when you use the add, remove, and replace APIs.
- Enabling security adds some overhead to the `init` or `open` APIs and other operations that work with documents inside the collection. Consider whether security is genuinely required. For example, the open API is much slower with encryption because it must generate the encryption keys that are used for encryption and decryption.
- The `replace` and `remove` APIs depend on the collection size as they must go through the whole collection to replace or remove all occurrences. Because it must go through each record, it must decrypt every one of them, which makes it much slower when encryption is used. This performance hit is more noticeable on large collections.
- The `count` API is relatively expensive. However, you can keep a variable that keeps the count for that collection. Update it every time that you store or remove things from the collection.
- The `find` APIs (`find`, `findAll`, and `findById`) are affected by encryption, since they must decrypt every document to see whether it is a match or not. For find by query, if a limit is passed, it is potentially faster as it stops when it reaches the limit of results. JSONStore does not need to decrypt the rest of the documents to figure out if any other search results remain.

Concurrency

JavaScript

Most of the operations that can be performed on a collection, such as add and find, are asynchronous. These operations return a jQuery promise that is resolved when the operation completes successfully and rejected when a failure occurs. These promises are similar to success and failure callbacks.

A jQuery Deferred is a promise that can be resolved or rejected. The following examples are not specific to JSONStore, but are intended to help you understand their usage in general.

Instead of promises and callbacks, you can also listen to JSONStore `success` and `failure` events. Perform actions that are based on the arguments that are passed to the event listener.

Example promise definition

```
var asyncOperation = function () {
  // Assumes that you have jQuery defined via $ in the environment
  var deferred = $.Deferred();

  setTimeout(function() {
    deferred.resolve('Hello');
  }, 1000);

  return deferred.promise();
};
```

Example promise usage

```
// The function that is passed to .then is executed after 1000 ms.
asyncOperation.then(function (response) {
  // response = 'Hello'
});
```

Example callback definition

```
var asyncOperation = function (callback) {
  setTimeout(function() {
    callback('Hello');
  }, 1000);
};
```

Example callback usage

```
// The function that is passed to asyncOperation is executed after 1000 ms.
asyncOperation(function (response) {
  // response = 'Hello'
});
```

Example events

```
$(document.body).on('WL/JSONSTORE/SUCCESS', function (evt, data, src, collectionName) {

  // evt - Contains information about the event
  // data - Data that is sent ater the operation (add, find, etc.) finished
  // src - Name of the operation (add, find, push, etc.)
  // collectionName - Name of the collection
});
```

Objective-C

When you use the Native iOS API for JSONStore, all operations are added to a synchronous dispatch queue. This behavior ensures that operations that touch the store are executed in order on a thread that is not the main thread. For more information, see the Apple documentation at Grand Central Dispatch (GCD) (https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html#//apple_ref/c/func/dispatch_sync).

Java

When you use the Native Android API for JSONStore, all operations are executed on the main thread. You must create threads or use thread pools to have asynchronous behavior. All store operations are thread-safe.

Analytics

ou can collect key pieces of analytics information that are related to JSONStore

File information

File information is collected once per application session if the JSONStore API is called with the analytics flag set to **true**. An application session is defined as loading the application into memory and removing it from memory. You can use this information to determine how much space is being used by JSONStore content in the application.

Performance metrics

Performance metrics are collected every time a JSONStore API is called with information about the start and end times of an operation. You can use this information to determine how much time various operations take in milliseconds.

Examples

iOS

```
JSONStoreOpenOptions* options = [JSONStoreOpenOptions new];
[options setAnalytics:YES];

[[JSONStore sharedInstance] openCollections:@[...] withOptions:options error:nil];
```

Android

```
JSONStoreInitOptions initOptions = new JSONStoreInitOptions();
initOptions.setAnalytics(true);

WLJSONStore.getInstance(...).openCollections(..., initOptions);
```

JavaScript

```
var options = {
  analytics : true
};

WL.JSONStore.init(..., options);
```

Working With External Data

You can work with external data in several different concepts: **Pull** and **Push**.

Pull

Many systems use the term pull to refer to getting data from an external source.
There are three important pieces:

External Data Source

This source can be a database, a REST or SOAP API, or many others. The only requirement is that it must be accessible from either the MobileFirst Server or directly from the client application. Ideally, you want this source to return data in JSON format.

Transport Layer

This source is how you get data from the external source into your internal source, a JSONStore collection inside the store. One alternative is a MobileFirst adapter.

Internal Data Source API

This source is the JSONStore APIs that you can use to add JSON data to a collection.

Note: You can populate the internal store with data that is read from a file, an input field, or hardcoded data in a variable. It does not have to come exclusively from an external source that requires network communication.

All of the following code examples are written in pseudocode that looks similar to JavaScript.

Note: Use MobileFirst adapters for the Transport Layer. Some of the advantages of using MobileFirst adapters are XML to JSON, security, filtering, and decoupling of server-side code and client-side code.

External Data Source: Backend REST endpoint
Imagine that you have a REST endpoint that read data from a database and returns it as an array of JSON objects.

```
app.get('/people', function (req, res) {

  var people = database.getAll('people');

  res.json(people);
});
```

The data that is returned can look like the following example:

```
[{id: 0, name: 'carlos', ssn: '111-22-3333'},
 {id: 1, name: 'mike', ssn: '111-44-3333'},
 {id: 2, name: 'dgonz' ssn: '111-55-3333')]
```

Transport Layer: MobileFirst adapter
Imagine that you created an adapter that is called people and you defined a procedure that is called getPeople. The procedure calls the REST endpoint and returns the array of JSON objects to the client. You might want to do more work here, for example, return only a subset of the data to the client.

```
function getPeople () {

  var input = {
    method : 'get',
    path : '/people'
  };

  return MFP.Server.invokeHttp(input);
}
```

On the client, you can use the WLResourceRequest API to get the data. Additionally, you might want to pass some parameters from the client to the MobileFirst adapter. One example is a date with the last time that the client got new data from the external source through the MobileFirst adapter.

```
var adapter = 'people';
var procedure = 'getPeople';

var resource = new WLResourceRequest('/adapters' + '/' + adapter + '/' + procedure, WLResourceRequest.GET);
resource.send()
.then(function (responseFromAdapter) {
  // ...
});
```

Note: You might want to take advantage of the `compressResponse`, `timeout`, and other parameters that can be passed to the `WLResourceRequest` API. Alternatively, you can skip the MobileFirst adapter and use something like jQuery.ajax to directly contact the REST endpoint with the data that you want to store.

```
$.ajax({
  type: 'GET',
  url: 'http://example.org/people',
})
.then(function (responseFromEndpoint) {
  // ...
});
```

Internal Data Source API: JSONStore After you have the response from the backend, you can work with that data by using JSONStore. JSONStore provides a way to track local changes. It enables some APIs to mark documents as dirty. The API records the last operation that was performed on the document, and when the document was marked as dirty. You can then use this information to implement features like data synchronization.

The change API takes the data and some options:

replaceCriteria
These search fields are part of the input data. They are used to locate documents that are already inside a collection. For example, if you select:

```
['id', 'ssn']
```

as the replace criteria, pass the following array as the input data:

```
[{id: 1, ssn: '111-22-3333', name: 'Carlos'}]
```

and the `people` collection already contains the following document:

```
{_id: 1,json: {id: 1, ssn: '111-22-3333', name: 'Carlitos'}}
```

The `change` operation locates a document that matches exactly the following query:

```
{id: 1, ssn: '111-22-3333'}
```

Then the `change` operation performs a replacement with the input data and the collection contains:

```
{_id: 1, json: {id:1, ssn: '111-22-3333', name: 'Carlos'}}
```

The name was changed from `Carlitos` to `Carlos`. If more than one document matches the replace criteria, then all documents that match are replaced with the respective input data.

addNew
When no documents match the replace criteria, the change API looks at the value of this flag. If the flag is set to **true**, the change API creates a new document and adds it to the store. Otherwise, no further action is taken.

markDirty
Determines whether the change API marks documents that are replaced or added as dirty.

An array of data is returned from the MobileFirst adapter:

```
.then(function (responseFromAdapter) {

  var accessor = WL.JSONStore.get('people');

  var data = responseFromAdapter.responseJSON;

  var changeOptions = {
    replaceCriteria : ['id', 'ssn'],
    addNew : true,
    markDirty : false
  };

  return accessor.change(data, changeOptions);
})

.then(function() {
  // ...
})
```

You can use other APIs to track changes to the local documents that are stored. Always get an accessor to the collection that you perform operations on.

```
var accessor = WL.JSONStore.get('people')
```

Then, you can add data (array of JSON objects) and decide whether you want it to be marked dirty or not. Typically, you want to set the markDirty flag to false when you get changes from the external source. Then, set the flag to true when you add data locally.

```
accessor.add(data, {markDirty: true})
```

You can also replace a document, and opt to mark the document with the replacements as dirty or not.

```
accessor.replace(doc, {markDirty: true})
```

Similarly, you can remove a document, and opt to mark the removal as dirty or not. Documents that are removed and marked dirty do not show up when you use the find API. However, they are still inside the collection until you use the `markClean` API, which physically removes the documents from the collection. If the document is not marked as dirty, it is physically removed from the collection.

```
accessor.remove(doc, {markDirty: true})
```

Push

Many systems use the term push to refer to sending data to an external source.

There are three important pieces:

Internal Data Source API

This source is the JSONStore API that returns documents with local-only changes (dirty).

Transport Layer

This source is how you want to contact the external data source to send the changes.

External Data Source

This source is typically a database, REST or SOAP endpoint, among others, that receives the updates that the client made to the data.

All of the following code examples are written in pseudocode that looks similar to JavaScript.

Note: Use MobileFirst adapters for the Transport Layer. Some of the advantages of using MobileFirst adapters are XML to JSON, security, filtering, and decoupling of server-side code and client-side code.

Internal Data Source API: JSONStore
After you have an accessor to the collection, you can call the `getAllDirty` API to get all documents that are marked as dirty. These documents have local-only changes that you want to send to the external data source through a transport layer.

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

accessor.getAllDirty()

.then(function (dirtyDocs) {
  // ...
});
```

The `dirtyDocs` argument looks like the following example:

```
[{_id: 1,
  json: {id: 1, ssn: '111-22-3333', name: 'Carlos'},
  _operation: 'add',
  _dirty: '1395774961,12902'}]
```

The fields are: * `_id`: Internal field that JSONStore uses. Every document is assigned a unique one. * `json`: The data that was stored. * `_operation`: The last operation that was performed on the document. Possible values are add, store, replace, and remove. * `_dirty`: A time stamp that is stored as a number to represent when the document was marked dirty.

Transport Layer: MobileFirst adapter
You can choose to send dirty documents to a MobileFirst adapter. Assume that you have a `people` adapter that is defined with an `updatePeople` procedure.

```
.then(function (dirtyDocs) {
  var adapter = 'people',
  procedure = 'updatePeople';

  var resource = new WLResourceRequest('/adapters/' + adapter + '/' + procedure, WLResourceRequest.GET)
  resource.setQueryParameter('params', [dirtyDocs]);
  return resource.send();
})

.then(function (responseFromAdapter) {
  // ...
})
```

Note: You might want to take advantage of the `compressResponse`, `timeout`, and other parameters that can be passed to the `WLResourceRequest` API.

On the MobileFirst Server, the adapter has the `updatePeople` procedure, which might look like the following example:

```
function updatePeople (dirtyDocs) {

  var input = {
    method : 'post',
    path : '/people',
    body: {
      contentType : 'application/json',
      content : JSON.stringify(dirtyDocs)
    }
  };

  return MFP.Server.invokeHttp(input);
}
```

Instead of relaying the output from the `getAllDirty` API on the client, you might have to update the payload to match a format that is expected by the backend. You might have to split the replacements, removals, and inclusions into separate backend API calls.

Alternatively, you can iterate over the `dirtyDocs` array and check the `_operation` field. Then, send replacements to one procedure, removals to another procedure, and inclusions to another procedure. The previous example sends all dirty documents in bulk to the MobileFirst adapter.

```
var len = dirtyDocs.length;
var arrayOfPromises = [];
var adapter = 'people';
var procedure = 'addPerson';
var resource;

while (len--) {

  var currentDirtyDoc = dirtyDocs[len];

  switch (currentDirtyDoc._operation) {

    case 'add':
    case 'store':

    resource = new WLResourceRequest('/adapters/people/addPerson', WLResourceRequest.GET);
    resource.setQueryParameter('params', [currentDirtyDoc]);

      arrayOfPromises.push(resource.send());

    break;

    case 'replace':
    case 'refresh':

    resource = new WLResourceRequest('/adapters/people/replacePerson', WLResourceRequest.GET);
    resource.setQueryParameter('params', [currentDirtyDoc]);


      arrayOfPromises.push(resource.send());

    break;

    case 'remove':
    case 'erase':

    resource = new WLResourceRequest('/adapters/people/removePerson', WLResourceRequest.GET);
    resource.setQueryParameter('params', [currentDirtyDoc]);

      arrayOfPromises.push(resource.send());
  }
}

$.when.apply(this, arrayOfPromises)
.then(function () {
  var len = arguments.length;

  while (len--) {
    // Look at the responses in arguments[len]
  }
});
```

Alternatively, you can skip the MobileFirst adapter and contact the REST endpoint directly.

```
.then(function (dirtyDocs) {

  return $.ajax({
    type: 'POST',
    url: 'http://example.org/updatePeople',
    data: dirtyDocs
  });
})


.then(function (responseFromEndpoint) {
  // ...
});
```

External Data Source: Backend REST endpoint
The backend accepts or rejects changes, and then relays a response back to the client. After the client looks at the response, it can pass documents that were updated to the markClean API.

```
.then(function (responseFromAdapter) {

  if (responseFromAdapter is successful) {
    WL.JSONStore.get('people').markClean(dirtyDocs);
  }
})


.then(function () {
  // ...
})
```

After documents are marked as clean, they do not show up in the output from the `getAllDirty` API.

Troubleshooting

For more information, see the JSONStore troubleshooting (../../troubleshooting/jsonstore) section.

API Usage

Select a platform:

Last modified on

IBM

Legal notices
(file:////home/travis/build/MFPSamples/DevCenter/_site/legal-notices/)
Privacy
(http://www.ibm.com/privacy/us/en/)
Terms of use
(file:////home/travis/build/MFPSamples/DevCenter/_site/terms-of-use/)
Third party notice
(file:////home/travis/build/MFPSamples/DevCenter/_site/third-party-notice/)

Social

Facebook
(https://www.facebook.com/ibmmobilefirstplatform/)
Twitter
(https://twitter.com/ibmmobiledev)
YouTube
(https://www.youtube.com/channel/UC1-C73-Aj4aKznci2Qusu97Q)
GitHub
(https://github.com/MobileFirst-Platform-Developer-Center)

Site

RSS feed
(file:////home/travis/build/MFPSamples/DevCenter/_s
Open issue
(https://github.com/MobileFirst-Platform-Developer-Center/DevCenter/issues/new)
Contribute
(https://github.com/MobileFirst-Platform-Developer-Center/DevCenter/blob/master/contributing.md)
Report abuse
(https://www.ibm.com/developerworks/community/re