iTunes Store
MZBookkeeper API Specification

Apple Confidential

Document Authors:

Garrick McFarlane	mcfarlane.g@apple.com	+447825746068 (London/Europe)
Chris Boulter	cboulter@apple.com	+442032846065 (London/Europe)

History:

1.0/d3	11 May 2010	First release version.
2.0/d6	29 Nov 2011	Updated to reflect latest API and functionality (mainly new MZBookkeeperClient class), and to set context relative to other Jingle KVS services.
d7	14 Dec 2011	Removed 1194='put refused due to size cap' status, now that oversize puts are silently discarded
d8	30 Jan 2012	Added information about 'getAll' with no keys.
d9	31 Jan 2012	Added domain-version to all API functions
d10	10 Feb 2012	Added punting info. Specified per-domain endpoint overrides. Slight tidy-up.

Contents:

Introduction

MZBookkeeper is the iTunes Store's internet-facing distributed key-value store. To decompose this definition:

- **iTunes Store.** MZBookkeeper is available for use by clients that have an authenticated iTunes user, identified by a DSID. Clients could be hardware devices e.g. the Apple TV or software products e.g. the iTunes application on Mac OS/Win/iOS. An authenticated DSID is needed before any data can be stored or retrieved; all data is stored against that DSID.
- **Internet-facing.** MZBookkeeper is intended to store data uploaded/downloaded by clients over the internet. For storing data generated/consumed internally on the server side, other key-value store APIs exist in the iTunes Store (Jingle) codebase. However, in addition to the internet-facing API, there is a client API available for embedding in other Jingle apps. Both APIs are documented here.
- **Distributed.** MZBookkeeper provides storage in a multi-data centre, replicated, very high performance database.
- **Key-value store.** MZBookkeeper's store acts as a distributed map (aka dictionary or associative array). Each DSID has its own namespace of keys; arbitrary blobs of data may be stored against the keys.

This document describes general Bookkeeper concepts, and provides a detailed specification of the two public Bookkeeper APIs:

- The internet client (HTTP) API
- The server-side (Java) API.

Examples of some current uses of the internet client API include:

- Apple TV playback positions, favourites and wishlists;
- iBooks and iTunes U reading positions, bookmarks and annotations;
- Apple TV playback performance data.

This document is mainly geared towards users of the internet client API.

Using MZBookkeeper

For further help with using MZBookkeeper, including getting new domains created for data storage, please contact the authors of this document.

Namespacing

There are three levels to the namespacing of an entry stored by MZBookkeeper.

- The first is the **account**, and it's implicit — it's specified by the validated/authenticated DSID as described above. There is no need to include the DSID within the key since all entries are automatically namespaced within a DSID by MZBookkeeper, but clients may choose to include the DSID to help them manage and generate keys.
- The second level is what's called the "**domain**" — this is a reverse-DNS–style string. This would typically identify your application, but may be shared between multiple client applications where applicable. An example would be "com.apple.ibooks", which would keep all the 'bookmark syncing' data inside a single logical namespace.
- The final level is the "**key**" itself. The format of the key is entirely arbitrary and is owned by the client. You would typically provide your own namespacing within the construction of your keys.

So in the case of storing an item of data about a user's book state, you might use one of the following key formats:

Example 1. [dsid].[book-id].[type] — eg, "`1241231.32124.annotations`"
Alternative: Example 2. [book-id].[type] — eg, "`32124.annotations`"

(Of course, you will need to choose and stick to one format for keys in your domain; the two examples above show keys with and without embedded DSIDs.)

Data Retention Policies (including LRU eviction)

Each domain can be configured to support its own LRU eviction and data throttling/capping policies. These attributes can help to control the volume of stored data for the domain. These are configured on the MZBookkeeper server-side. The appropriate policies for your domain should be agreed with the Bookkeeper owners as part of your domain bring-up.

Getting HTTP endpoints from the Store Bag

The API for internet clients comprises the following calls—

- get
- getAll
- put
- putAll *and*
- lock.

All calls are HTTP POST over SSL with the call identified by the last part of the URL path. Payloads are XML Property List.

The endpoints to use for the API are provided via the keybag, under the following default keys which may be overridden per domain as described shortly:

- `kvs-get`
- `kvs-getall`
- `kvs-put`
- `kvs-putAll` *and*
- `kvs-lock`

If the keys are not present in the bag, the client *must* consider the service to be unavailable at that time. We reserve the right to remove the keybag entries, eg during maintenance or an outage.

Keybag lookup for API endpoints

The keybag will contain default endpoints (except eg during an outage) and may also contain per-domain overrides. Clients MUST look for per-domain overrides[1], and use these as the destination when sending requests for a domain with overrides. Clients are to fall back to the default endpoints only when no overrides are present for the domain in question. Example bag entries are below:

```
<key>kvs-get</key><string>http://itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/get</string>
<key>kvs-getall</key><string>http://itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/getAll</string>
<key>kvs-put</key><string>http://itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/put</string>
<key>kvs-putall</key><string>http://itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/putAll</string>
<key>kvs-lock</key><string>http://itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/lock</string>

<key>kvs-get.com.apple.upp</key><string>http://blah.itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/get</string>
<key>kvs-getall.com.apple.upp</key><string>http://blah.itunes.apple.com/WebObjects
```

[1] Per-domain overrides are being introduced from Q1 2012. We will maintain backward compatibility for legacy clients created before this mechanism was introduced, but need new clients to honour the per-domain overrides. We require support for this by clients using domains created from Q1 2012 onwards, including 'com.apple.upp'.

```
/MZBookkeeper.woa/wa/getAll</string>
<key>kvs-put.com.apple.upp</key><string>http://blah.itunes.apple.com/WebObjects/MZ
Bookkeeper.woa/wa/put</string>
<key>kvs-putall.com.apple.upp</key><string>http://blah.itunes.apple.com/WebObjects
/MZBookkeeper.woa/wa/putAll</string>
<key>kvs-lock.com.apple.upp</key><string>http://blah.itunes.apple.com/WebObjects/M
ZBookkeeper.woa/wa/lock</string>
```

When per-domain overrides are in use, they will always be indicated by adding a suffix to the default keys. The suffix is made up of a period then the domain name, as shown – so 'kvs-get' is overridden by 'kvs-get.com.apple.upp'.

Authentication

All invocations of the internet API must be authenticated. That means that each HTTP transaction must include appropriate headers to supply the DSID and matching token to prove that the user has authenticated themselves. The server validates this pair on each call. If the token has expired, the call is rejected with an authentication failure.

If the dsid/token pair are successfully validated, the transaction proceeds. Entries on the server are transparently namespaced/sandboxed within the validated dsid — **you cannot retrieve a value stored by one account with authentication headers for another account.**

Authentication Technical Details

Whenever you authenticate the client (by invoking `https://buy.itunes.apple.com/WebObjects/MZFinance.woa/wa/authenticate`, such as happens during a sign-in, whether manual or triggered through a purchase attempt), Jingle's response contains the following Set-Cookie headers of interest here:

```
Set-Cookie:
X-Dsid=1046272497; version="1"; expires=Sat, 21-Aug-2010 17:01:45 GMT;
path=/WebObjects; domain=.apple.com,
mz_at0=AQQBAAFekAABAACKjtL5AcVKDGZ/w7GtYqYMbnwheaa/vq0=; version="1"; expires=Fri,
21-Aug-2009 17:31:45 GMT; path=/WebObjects; domain=.apple.com
```

(also `Pod=1; version="1"; expires=Mon, 21-Sep-2009 17:01:45 GMT; path=/; domain=.apple.com`)

All your subsequent MZBookkeeper requests against that account will then need to include these same cookies, for the request to be allowed. (You should also be sending the same `X-Dsid` as an HTTP header.)

The `mz_at0` cookie is the 'authentication token'. It will be valid for 180 days after being issued. Its expiry/validity is checked on every request on the server-side, so it is not strictly speaking essential that you manage its expiry on the client. Moreover, you should be sure *not* to discard it in fewer than 180 days, unless perhaps you choose to do so when the customer signs out from the store.

The authentication token is also known as a 'weak token'. Weak Token Authentication is described in the document *Weak Token Authentication Usage Primer* available from the authors of this document.

If a Key-Value Store request is considered to be not sufficiently authenticated on the server-side, we will return an 'authentication error' status (-4) in the response XML Property List. We do *not* return an iTunes protocol response, nor do we issue a redirect to force an authentication at that point. (This is different from the mechanics of what currently happens during a purchase, for example.) The client should handle the receipt of a 'not authenticated' response with appropriate UI or other measures to take the customer through a fresh authentication.

If your requests are split between multiple processes, such as the `itunestored` and your own front end app, these would typically need to share the same cookie jar for the above process to work.

If the customer changes their iTunes password (through any means, from any location, not necessarily the same device that the client is running on), the server considers the previously issued token as invalidated, and will no longer accept it.

Version Tracking Support

This section describes Bookkeeper's *key*-level versioning; in early 2012, we plan to add *domain*-level versioning. Domain-level versioning is not yet described in this section of this document, but is shown in the Internet Client API Specification (next section).

The API includes optional 'version tracking' support — really a simple optimistic locking mechanism to identify where an entry has been updated on the server behind the client's back. This mechanism supports two functions:

1. use cases such as last-read page syncing in iBooks. The customer may be reading the same book across multiple devices, and we need a way to handle conflicts in the page number data from their different devices.
2. reducing network traffic on 'get' requests via a mechanism very similar to HTTP 'If-Modified-Since'.

You can choose to ignore this version tracking support altogether, in which case all writes are blind and operate as last-write-wins. For some lightweight use cases, this is a sensible and efficient strategy. To take this approach, omit version specifier information from your put calls (see later), and none will be returned in the responses.

Where you need to detect and/or prevent data loss due to multiple writers attempting to put data to the same entry on the server, you should take advantage of the version-tracking support. To do this on a `put`, you specify the client's last-known-version in the `put` parameters. If the server finds that the entry on the server has already been updated since then (ie, the server version identifier for that key is different to the one specified by the client), then instead of overwriting the stored data with the new value, the server returns its own latest known state and accompanying version identifier, so that the client can

a. apply whatever conflict resolution strategy it chooses; and then
b. re-try the `put` with the new version identifier.

For a successful `put` supplied with versioning information, the response will always include a new version identifier that the client can track for any subsequent puts against the same key.

A `get` will always return a version identifier along with the current value.

Note that version identifiers are currently implemented in MZBookkeeper as integers, but this is not part of the API specification and may change. Clients should treat MZBookkeeper identifiers as they would GUIDs or any other opaque blobs. This means you may test for presence or equality, but you must not compare version identifiers (eg, using the < or > operators) or mutate a version identifier (eg, ++).

First Version-Tracked Write

The version tracking mechanism described above does not cater for the situation where you are making your first `put` from a client, for a given key, and your client has no previous known state about that key. You might want the server to reject the put if it already has a value stored for that key, but in order to do that you need to provide a `base-version` in your put call. If you have never previously interacted with the server for that key, you will not have a candidate `base-version` to use at that point.

To address this situation, there is an agreed special-case `base-version` of "1" which you can use in an initial `put`. This is always treated as a version-mismatch for that key by the server, which will thus return the current value and version identifier. The client can then resolve as necessary, then try a further `put` with the new information, which should succeed. (If the key did *not* already exist in the first place, then the initial `put` would succeed, and the new version identifier would be returned in the response.)

Version field naming

Note that:

- versioned `put` and `putAll` requests should specify `base-version`, whereas
- versioned `get` and `getAll` requests should specify `since-version`

(Historically some clients have confused these two field names and MZBookkeeper has not validated this. Using the correct names clarifies how versioning works in your client.)

Request punting

Request punting is a mechanism by which Bookkeeper may refuse to process a request due to excessive load. Any request to Bookkeeper may result in a response like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

<dict>

  <key>retry-seconds</key><integer>120</integer>

  <key>status</key><integer>1197</integer>

</dict>
```

```
</plist>
```

The 1197 status code denotes a punted request. When this response is seen, clients should fail gracefully where possible. They may retry their request after the indicated ‘retry-seconds’. Ideally, clients should not send other Bookkeeper requests during the period up to the expiry of ‘retry-seconds’.

Internet Client API Specification

Status codes in responses

All MZBookkeeper responses include a status code which may be interpreted using the following list. Note when dealing with putAll/getAll requests that the status code is per response, not per entry.

Status code	Description	Applies to
-1	Initialization error	*Not currently used*
-2	Validation error The request sent by the client was invalid, eg, it referenced a non-existent domain.	All requests
-3	Generic error An unspecified server-side error was encountered; this may be reported to the MZBookkeeper team	All requests
-4	Authentication error	All requests
0	Success	All requests
1101	Unsupported client The client is not supported (based on its User-Agent header or other details)	*Not currently used*
1199	Unspecified error An unspecified server-side error was encountered; this may be reported to the MZBookkeeper team	All requests
1198	Version mismatch See earlier in this document (Version Tracking Support)	put putAll (possible for get/getAll but shouldn't be seen unless clients ignore the instruction to avoid mutating version identifiers)
1197	Request punted The server was unable to process the request due to a temporary overloading. The implication is that this is a temporary condition which will be alleviated after some delay.	All requests
1196	Account locked See later in this document	get put putAll lock (nb, getAll is permitted for locked accounts)
1195	Account put throttled The put (or one or more of the puts in a putAll) was refused due to too many puts being received for this DSID for this domain in a configured time period	put putAll

`get`: Retrieve a single value from the store

This looks up and returns the current value for a key from the store.

This is optimised for single-key lookup. To retrieve multiple items efficiently, see 'getAll' below.

get			
Parameter name	**Cardinality**	**Data type**	**Notes**
Client-generated Request →			
domain →	1	String (255)	Used to qualify the namespace of the key, to avoid unintended collisions. Although the format is not enforced, it is recommended to use a Reverse-DNS format — for example: "com.apple.ibooks".
key →	1	String (255)	The key to look up.
			Note that the following two parameters (*since-version* and *since-domain-version*) are optional optimisations to reduce response traffic where the client is polling for server-side updates to a value already held on the client (i.e. for updates made by other clients). This applies to *get* though not to *getAll* (next section). Note also that it is an error to specify both *since-version* and *since-domain-version* in a request; at most one of these should be specified. If neither of these is present in the request, the found entry is always returned.
since-version →	0..1	String (255)	Only return the found entry if the version held in the store is newer than *since-version*.
since-domain-version →	0..1	Number (Integer)	Only return the found entry if the version held in the store has been put (created or mutated) after the store was at the specified domain-version.
← Store–generated Response			
← status	1	Number (Integer)	OSStatus-style integer indicating success or otherwise of the operation. See table earlier.
← domain-version	1	Number (Integer)	The current domain-version for the DSID in the domain.
← value	1	Data	The retrieved value.
← version	1	String (255)	The current version identifier of the entry on the server. This should be treated as opaque by clients.

Example

This example retrieves the value for keys `9441029.5512345.position` for DSID 9441029.

Request:

Note this example URL came from the bag for a user on Pod 8; clients *must* use the bag mechanism to determine the correct URL.

```
POST https://p8-buy.itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/get HTTP/1.1
```

HTTP Headers —

```
Content-Type: text/xml; charset=UTF-8

X-Dsid: 9441029

Cookie: ...etc...
```

POST Body —

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

            <dict>

                    <key>domain</key>

                    <string>com.apple.ibooks</string>

                    <key>key</key>

                    <string>9441029.5512345.position</string>

                    <key>since-version</key>

                    <string>00010000010000012805232321</string>

            </dict>

</plist>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">


<plist version="1.0">

            <dict>

                    <key>status</key>

                    <integer>0</integer>

                    <key>domain-version</key>

                    <integer>12345</integer>

                    <key>value</key>

                    <data>A2FiCg==</data>

                    <key>version</key>

                    <string>00010000010000012805483234</string>
```

```
            </dict>
        </plist>
```

`getAll`: Retrieve multiple values from the store at once

This call can be used to retrieve multiple key/value pairs in a single invocation. It is identical to the `get` call detailed above, except that it takes and returns an array of keys at once. You may omit the array of keys from the request in order to retrieve all entries held for the account, and this may be a common usage when a domain-version is specified, in order to retrieve all entries put (created/mutated) since the store was at a particular domain-version. If you pass it a one-element array of keys, it has exactly the behaviour of the single-item `get` call, although is slightly less efficient due to its verbosity.

getAll

Parameter name	Cardinality	Data type	Notes
		Client-generated Request →	
domain →	1	String (255)	Used to qualify the namespace of the key, to avoid unintended collisions. Although the format is not enforced, it is recommended to use a Reverse-DNS format — for example: "com.apple.ibooks". Note that all keys in a given invocation must be within the same domain.
since-domain-version →	0..1	Number (Integer)	If present in the request, this will filter the response to include only entries which were put into the store (created or mutated) after the store was at the specified-domain version.
list ("keys") 1..n —			List of keys to return. Omit to return all keys held for the account (or all keys which have been created/mutated since the store was at a particular domain-version, if *since-domain-version* is specified).
key →	1	String (255)	A key to look up.
since-version →	0..1	String (255)	Only return the found entry if the version held in the store is newer than *since-version*. This is an optional optimisation to reduce response traffic where the client is polling for server-side updates to values already held on the client. If since-version is omitted from the request, the found entry is always returned. Note that it is an error to specify *since-version* for any of your requested keys if you are also specifying a *since-domain-version* for the request; at most one of these elements should be in use for any given request.
		← Store–generated Response	
← status	1	Number (Integer)	OSStatus-style integer indicating success or otherwise of the operation. See table earlier.
← domain-version	1	Number (Integer)	The current domain-version for the DSID in the domain.
list ("values") 1..n —			
← key	1	String (255)	Key for this entry in the values list.
← value	1	Data	Retrieved value for this entry in the list.
← version	1	String (255)	The current version identifier of this entry on the server. This should be treated as opaque by clients.

Example

This example retrieves the values for keys `9441029.5512345.position` and `9441029.5512345.annotations`.

Request:

Note this example URL came from the bag for a user on Pod 8; clients *must* use the bag mechanism to determine the correct URL.

```
POST
https://p8-buy.itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/getAll HTTP/1.1
```

Headers—

```
Content-Type: text/xml; charset=UTF-8

X-Dsid: 9441029

Cookie: ...etc...
```

POST Body —

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

           <dict>

                  <key>domain</key>

                  <string>com.apple.ibooks</string>

                  <key>keys</key>

                  <array>

                         <dict>

                                <key>key</key>

                                <string>9441029.5512345.position</string>

                                <key>since-version</key>

                                <string>00010000010000012805232321</string>

                         </dict>

                         <dict>

                                <key>key</key>

                                <string>9441029.5512345.annotations</string>

                                <key>since-version</key>

                                <string>00010000010000012804231221</string>

                         </dict>

                  </array>

           </dict>
```

```
</plist>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">
	<dict>
		<key>status</key>
		<integer>0</integer>
		<key>domain-version</key>
		<integer>12345</integer>
		<key>values</key>
		<array>
			<dict>
				<key>key</key>
				<string>9441029.5512345.position</string>
				<key>value</key>
				<data>A2FiCg==</data>
				<key>version</key>
				<string>00010000010000012805483234</string>
			</dict>
			<dict>
				<key>key</key>
				<string>9441029.5512345.annotations</string>
				<key>value</key>
				<data>NFAwgWtTxchp0 … bKutTxchp0F6HbK==</data>
				<key>version</key>
				<string>00010000010000012804231221</string>
			</dict>
		</array>
	</dict>
</plist>
```

`put`: Write a single value to the store

This call is optimised for writing a single value to the store. It supports optional versioning, whereby the entry is written only if the version of the entry held on the server matches the version supplied by the client. In the case where a mismatch is detected, the current state is returned. If versioning is not used, the entry is written regardless of current server-side state, and no state information is returned.

put			
Parameter name	**Cardinality**	**Data type**	**Notes**
Client-generated Request →			
domain →	1	String (255)	Used to qualify the namespace of the key, to avoid unintended collisions. Although the format is not enforced, it is recommended to use a Reverse-DNS format — for example: "com.apple.ibooks". Note that all keys in a given invocation must be within the same domain.
key →	1	String (255)	The key to write to.
value →	1	Data	The data to store for the key.
base-version →	0..1	String (255)	Only write the entry if the current server version for the key matches base-version. If this optional value is not specified, then the entry is always written. If base-version is specified, and it matches the current version of the entry on the server, the value is stored and a new version identifier is generated and returned. If base-version is specified but does not match the current version identifier of the entry on the server, then the value is not stored, and the server's current entry is returned as though a 'get' were performed. This allows the client to perform local conflict resolution, updating its view of the server state, and retry the put with an up-to-date version.
← Store–generated Response			
← status	1	Number (Integer)	OSStatus-style integer indicating success or otherwise of the operation. See table earlier.
← domain-version	1	Number (Integer)	The current domain-version for the DSID in the domain.
← value	0..1	Data	Current value for the entry on the server. This is returned only in the case of a version mismatch.
← version	0..1	String (255)	The current or new version identifier of the entry on the server. For a successful, versioned put (ie, with base-version specified in the input), this reflects the new state of the entry on the server. For an unsuccessful versioned put (due to a versioning conflict), this reflects the newest version of the entry on the server. Non-versioned puts (ie, without base-version specified in the input) do not return the new version identifier.

Example

This example performs a non-versioned put for key `9441029.5512345.position` to the store. The write is successful, and no new version identifier is returned.

Request:

Note this example URL came from the bag for a user on Pod 8; clients *must* use the bag mechanism to determine the correct URL.

```
POST https://p8-buy.itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/put HTTP/1.1
```

Headers—

```
Content-Type: text/xml; charset=UTF-8

X-Dsid: 9441029

Cookie: ...etc...
```

POST Body —

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

            <dict>

                    <key>domain</key>

                    <string>com.apple.ibooks</string>

                    <key>key</key>

                    <string>9441029.5512345.position</string>

                    <key>value</key>

                    <data>A2FiCg==</data>

            </dict>

</plist>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

            <dict>

                    <key>status</key>

                    <integer>0</integer>

                    <key>domain-version</key>

                    <integer>12345</integer>

            </dict>

</plist>
```

`putAll`: Write multiple values to the store at once

This call can be used to write multiple entries to the store in a single invocation. Each entry may use versioning or not, and the response will contain conflict data accordingly for each item. Note that in this case, some items may be written and some not — *ie* transaction granularity is per-item, not per invocation.

putAll			
Parameter name	**Cardinality**	**Data type**	**Notes**
		Client-generated Request →	
domain →	1	String (255)	Used to qualify the namespace of the key, to avoid unintended collisions. Although the format is not enforced, it is recommended to use a Reverse-DNS format — for example: "com.apple.ibooks". Note that all keys in a given invocation must be within the same domain.
list ("keys") 1..n —			
***key* →**	1	String (255)	The key to write to.
***value* →**	1	Data	The data to store for the key.
***base-version* →**	0..1	String (255)	Only write the entry if the current server version for the key matches base-version. If this optional value is not specified, then the entry is always written. If base-version is specified, and it matches the current version of the entry on the server, the value is stored and a new version identifier is generated and returned. If base-version is specified but does not match the current version identifier of the entry on the server, then the value is not stored, and the server's current entry is returned as though a 'get' were performed. This allows the client to perform local conflict resolution, updating its view of the server state, and retry the put with an up-to-date version.
		← Store–generated Response	
← status	1	Number (Integer)	OSStatus-style integer indicating success or otherwise of the operation. See table earlier.
← domain-version	1	Number (Integer)	The current domain-version for the DSID in the domain.
list ("values") 1..n —			
← *key*	1	String (255)	Key for this entry in the values list.
← *value*	0..1	Data	Current value for the entry on the server. This is returned only in the case of a version mismatch.
← *version*	0..1	String (255)	The current or new version identifier of the entry on the server. For a successful, versioned put (ie, with base-version specified in the input), this reflects the new state of the entry on the server. For an unsuccessful versioned put (due to a versioning conflict), this reflects the newest version of the entry on the server. Non-versioned puts (ie, without base-version specified in the input) do not return the new version identifier.

Example

This example attempts to write three entries in a single round-trip. It puts values for the keys "**9441029.5512345.position**", "**9441029.5512345.annotations**", and "**9441029.252323412.coffee-stains**".

There is some extra complexity in this example —

- The "...`position`" write is non-versioned; the value passed in replaces the stored value on the server, no questions asked, and no versioning information is sent back in the response.
- The "...`annotations`" write is versioned, and fails due to the server having a more recent version stored. The response contains the newer value for this key as well as its version identifier.
- The "...`coffee-stains`" write is versioned, and succeeds. The response contains the newer version identifier, but no value.

Request:

Note this example URL came from the bag for a user on Pod 8; clients *must* use the bag mechanism to determine the correct URL.

```
POST https://itunes.apple.com/WebObjects/MZCumulus.woa/wa/putAll HTTP/1.1
```

Headers—

```
Content-Type: text/xml; charset=UTF-8

X-Dsid: 9441029

Cookie: ...etc...
```

POST Body —

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

            <dict>

                    <key>domain</key>

                    <string>com.apple.ibooks</string>

                    <key>keys</key>

                    <array>

                            <dict>

                                    <key>key</key>

                                    <string>9441029.5512345.position</string>

                                    <key>value</key>

                                    <data>A2FiCg==</data>

                            </dict>

                            <dict>
```

```
                                <key>key</key>
                                <string>9441029.5512345.annotations</string>
                                <key>value</key>
                                <data>C2FiCg==</data>
                                <key>base-version</key>
                                <string>00010000010000012874ea2323</string>
                        </dict>
                        <dict>
                                <key>key</key>
                        <string>9441029.252323412.coffee-stains</string>
                                <key>value</key>
            <data>J2R4fTNFAwgWtTxchp0F6HbKuWHu4F0L5KB…5R5KBBmLu9QQULA==</data>
                                <key>base-version</key>
                                <string>00010000010000012806a60882</string>
                        </dict>
                </array>
            </dict>
</plist>
```

Response:

```
<plist version="1.0">
            <dict>
                    <key>status</key>
                    <integer>0</integer>
                    <key>domain-version</key>
                    <integer>12345</integer>
                    <key>values</key>
                    <array>
                            <dict>
                                    <key>key</key>
                                    <string>9441029.5512345.annotations</string>
                                    <key>value</key>
                                    <data>kwAAAABK54kwAAAABK54kwAAAABK54==</data>
                                    <key>version</key>
                                    <string>00010000010000001289a4a6eea</string>
                            </dict>
```

```
                        <dict>
                                <key>key</key>
                        <string>9441029.252323412.coffee-stains</string>
                                <key>version</key>
                                <string>0001000001000012832eaad21</string>
                        </dict>
                </array>
        </dict>
</plist>
```

`lock`: Lock an iBooks account for iCloud migration

This is not part of the core MZBookkeeper API, but exists to serve a specific requirement: marking iBooks accounts which have been migrated from MZBookkeeper to iCloud. iBooks accounts are those which store MZBookkeeper data in the domain "`com.apple.ibooks`". Once an account has been locked, subsequent `get`, `put`, `putAll` and `lock` requests will be rejected with an 'account locked' status (see table of status codes earlier). `getAll` requests are still possible for locked accounts.

lock

Parameter name	Cardinality	Data type	Notes
Client-generated Request →			
domain →	1	String (255)	Used to specify the domain in which the account is to be locked. Although the format is not enforced, it is recommended to use a Reverse-DNS format — for example: "com.apple.ibooks".
← Store–generated Response			
← status	1	Number (Integer)	OSStatus-style integer indicating success or otherwise of the operation. See table earlier.
← domain-version	1	Number (Integer)	The current domain-version for the DSID in the domain.

Example

This example locks DSID 9441029.

Request:

Note this example URL came from the bag for a user on Pod 8; clients *must* use the bag mechanism to determine the correct URL.

```
POST https://p8-buy.itunes.apple.com/WebObjects/MZBookkeeper.woa/wa/lock HTTP/1.1
```

HTTP Headers —

```
Content-Type: text/xml; charset=UTF-8

X-Dsid: 9441029

Cookie: ...etc...
```

POST Body —

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

          <dict>

                <key>domain</key>
```

```
                <string>com.apple.ibooks</string>
        </dict>
</plist>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
        <dict>
                <key>status</key>
                <integer>0</integer>
                <key>domain-version</key>
                <integer>12345</integer>
        </dict>
</plist>
```

Serverside API

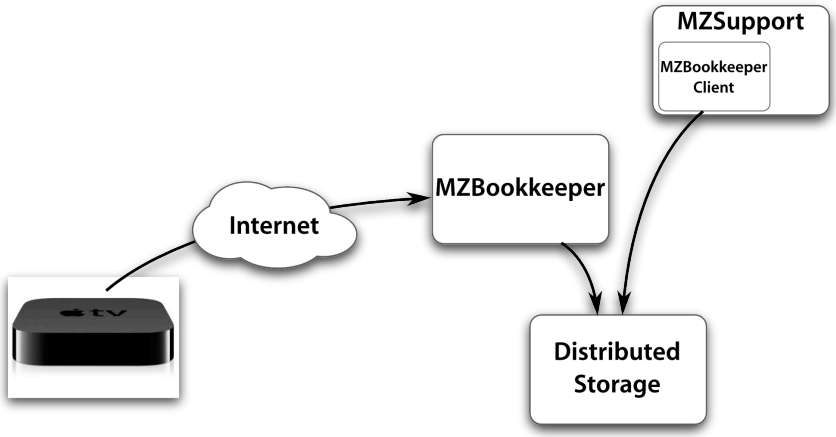
This section is intended for iTunes Store server-side engineers wishing to access Bookkeeper data internally. Client-side developers can skip this section entirely.

The internal MZBookkeeper client takes the form of a Java class, MZBookkeeperClient, which may be used by iTunes Store (Jingle) server-side applications to access data stored in MZBookkeeper domains.

MZBookkeeper is only intended for internet-accessible storage, so a typical deployment including MZBookkeeperClient may involve:

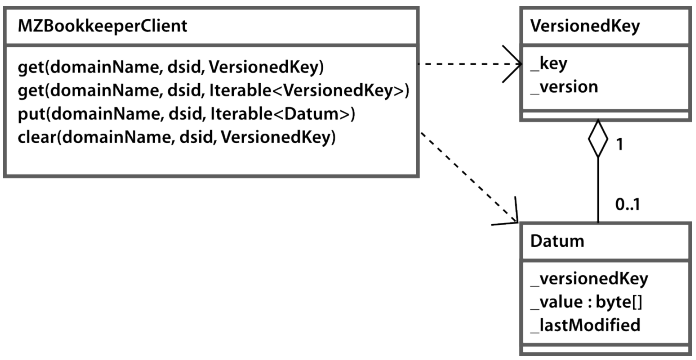
- hardware devices uploading data using MZBookkeeper's internet API
- Jingle apps accessing that data using MZBookkeeperClient

This is indeed how 'ATV diagnostics' files will be handled: the ATV boxes will upload their diagnostics files using MZBookkeeper's internet API described earlier, and MZSupport will provide access to these files using MZBookkeeperClient, as shown in the following diagram.



Classes

The API comprises the following classes.



The following methods are exposed by MZBookkeeperClient:

Method	Description
`get(String domain, Long dsid, VersionedKey requestedKey) : Datum`	
	Get a single entry from the store for a specified domain and DSID

`get(String domain, Long dsid, Iterable<? extends VersionedKey> requestedKeys) : Map<VersionedKey, Datum>`	
	Get multiple entries from the store for a specified domain and DSID
`put(String, Long, Iterable<? extends Datum>) : Map<VersionedKey, Datum>`	
	Put an entry into the specified domain for the specified DSID
`clear(String, Long, VersionedKey) : boolean`	
	Delete a specified entry from the specified domain for the specified DSID.

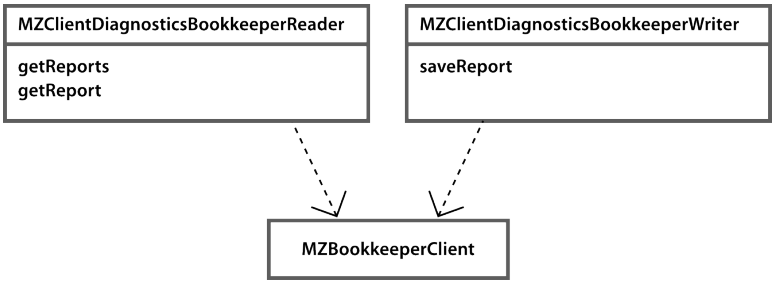
The other classes of interest are:

- **Datum.** POJO representing a key/value entry in the store. Encapsulates a VersionedKey.
- **VersionedKey.** POJO representing the key to an entry in the store. Note that the version field may be null for unversioned storage.

The API classes are reasonably well Javadoc'd.

Using the Java API

As a Jingle developer you should generally add a layer of abstraction around MZBookkeeperClient, using the vocabulary of your application to expose storage functions. For an example, see these two classes which abstract MZBookkeeperClient for use by ATV diagnostics ('Client Diagnostics'):



Differences from Internet Client API

Users of MZBookkeeperClient, the Java API, should be aware of the following differences from the Internet Client API documented previously:

- behaviour around returning keys of successfully written entries (the internet API returns these keys, whereas MZBookkeeperClient's put method doesn't)
- support for lockable stores (Java API users may see a special 'tombstone' entry returned for a locked account; the Java client should not be used for lockable stores until this is resolved)
- support for specifying a 'since-version' when calling the `get` method (this is intended to reduce network traffic in the internet API and is not needed for the Java API).
- possibly, differences around 'domain-version' (yet to be decided).
- authentication is not needed using the Java API

[end of document]