# A “refresh”ing way to save changes in an itemMirror object.

This document describes an auto-save mode for use of itemMirror methods — in particular for use of methods that change the state of an iM object (e.g., methods that set, create, delete, add, remove). With auto-save, performance when using

## The standard, “manual” way to save changes.

Let’s first review standard way to save changes in an itemMirror object. Assume:

* There is an itemMirror application. Call it “app”.
* The app, at some point in the current session, has constructed an itemMirror (iM) object. Call this *iM*.
* *iM* mirrors the structure of a grouping item such as a folder. Call this *F*.
* Under the folder model, the information in *iM,* including metadata (namespace) attributes is persisted as XML structured according to the XooML2 schema and saved within *F* as a “xooml2.xml” file.[[1]](#footnote-1)
* At some point, the app decides to .*save* changes. If *iM*.GUIDGeneratedOnLastWrite equals its counterpart in the xooml2.xml file, then the save completes as follows:

1. A new GUID is generated and the GUIDGeneratedOnLastWrite attribute in *iM* is set to this value.
2. Information in the *iM* object is serialized into XML.
3. Using the specified xoomlDriver, this XML replaces, entirely, the contents of the xooml2.xml file.

***Important!:*** *in some earlier documentation, .save was also described as including a “sync”. This is not needed for .save and should not be done.* ***Sync (i.e., private method .\_sync) should only be done with .refresh and initial construction of an iM object.***

But if the two GUIDGeneratedOnLastWrite attributes do NOT match then the *save* fails and a .*refresh* must be invoked before another attempt to *.save* can succeed. The specified versions of all three drivers, itemDriver, xoomlDriver and syncDriver, are involved to build a new version of *iM* following the same actions as are used to construct *iM* in the first place:

1. Through the specified xoomlDriver, retrieve the contents of the xooml2 representation for *F*. And use these to create a new version of *iM*
2. Through the specified itemDriver, retrieve a listing of the current contents for folder F.
3. Through the specified syncDriver, compare contents retrieved through itemDriver with those (esp. associations) in the new *iM* object. In cases of conflict, resolve in favor of the reality of contents for folder *F*. For example, if a file or subfolder is listed for folder *F* that is not represented by an association in *iM* then add a new association (with a new GUID id) to the *iM* object. If a real association (for file or subfolder) is in *iM* but not listed as a subfolder or file for folder *F*, then remove this association from the new *iM* object.
4. Assign a new GUID as value to *iM*’s attribute, GUIDGeneratedOnLastWrite.
5. This new version of *iM* reflects the latest in contents of the xooml2.xml file and the current reality of folder F. Serialize this object into XML and, using the xoomlDriver, save this XML back into the xooml2.xml file.
6. Note that any changes made to *iM* since the last successful save are lost. However, if the methods through which these changes were made have been buffered, then changes can be re-introduced to the new *iM*.
7. Now the .*save* can be attempted again. And this attempt is likely to succeed if attempted immediately after .*refresh* and the re-application of the changes made since the last successful save.

But if it fails? Then the application might try the whole cycle again (refresh, re-application of changes since last successful save, then attempt save again). And again… And again…

The problem with this approach is that it places considerable responsibility in the hands of the app developer: Changes (via methods) must be buffered. A refresh, re-application, re-try save must be built in as fallback to each save attempt.

## What about leading with *.refresh* with each application of a method that changes the iM object and then concluding with a forced save?

Changes an app might wish to save for an iM object can only be made through a small set of methods for the iM class. These include methods to “add” (a namespace attribute), “remove”, “create”, “delete”, “set” and “move”. What if, in their use of itemMirror, applications could opt to use an “auto-save” mode so that a save is forced with each application of one of these “changing” methods (i.e., any method that changes the state of the iM object)? Let’s pick one method, say .*setDisplayText.* Here is how this would work:

1. *.refresh …* that’s right! Do a refresh even before the method is actually applied. Normally, a refresh might mean that changes made since the last save are lost. But if all is done in auto-save mode, then each such change is saved immediately after it is made.
2. Execute the method to make the change.
3. .\_*saveForced.* Serialize the iM object into XML and save this to the appropriate xooml2.xml file without doing a check for matching GUIDs (in GUIDGeneratedOnLastWrite for the xooml2.xml file and for the *iM* object). (Question: .\_*saveForced*  is a new proposed private method — really just a variation on *.save* but with no check of GUIDGeneratedOnLastWrite. How does this method compare to the current . \_*unsafeWrite* method?)

Let’s look at the cons and then the pros of this “refresh”ing approach to persistence. Cons:

1. If several different iM objects (in different apps, run by different users) are targeting the same folder F (or some other grouping item) at the same time, then there is a small chance that the forced save may overwrite the save done through another iM object. This might happen, for example, if a sequence of .*refresh,* change method(s), .*save* is issued through another iM object (for the same folder F) in the small period of time somewhere ­­­­­\_in between­\_ the .refresh and .*saveForced* operations that bracket a change method when itemMirror is used in auto-save mode. In worst case, not only are changes made through the other iM object lost but also the change method for the current iM object may fail. (In no cases, however, should conflict lead to a corrupted iM object or a corrupted xooml2.xml file).
2. Bracketing each change method with *.refresh* and *.save* means more file operations and slower times to complete. How much slower is still TBD.

But then for a big pro:

1. App developers needn’t worry about the possibility of a failed save nor the possible need to buffer changes in case of failure nor the need to decide when to issue *.save.* All changes are immediate and automatic. The auto-save mode is especially useful for developers inexperienced with itemMirror. Developers can focus on use of the methods without worry about when to use *.save* and *.refresh.*

## Consider this approach for more complex methods that involve file system operations.

This auto-save mode (bracket a change method with an initial *.refresh* and a concluding *.\_saveForced*) may be the best way to handle more complex methods that involve file system operations.

Consider .*move* i.e., the move of an association (as specified by GUID) from *iM* (again representing folder *F*)to a destination iM object (e.g. representing a folder *F2*). Call this destination object *iM2* (the “itemMirror” parameter of .*move*). If the association is “real” (vs. “phantom”) then it represents a subfolder or file under folder *F.* So a *.move*  involves a real file system operation (as managed through the specified itemDriver). Nothing else (e.g., changes to xooml2 files) can happen until this operation completes successfully. (And there are many reasons a move of file or subfolder might fail including that the file or subfolder is “open”.)

Note: If app developers are not in auto-save mode, then their code should be sure to *.save* both the source and destination objects, *iM* and *iM2*, before invoking the *.move* method.

If the file system move succeeds then itemMirror needs to complete some mop-up operations as follows:

1. *.copyAssociaiton* (GUID) as applied to *iM.* Do this and NOT .*deleteAssociation* since the *.localItem* has already moved and the association will be automatically removed on the next application of .*\_sync* (e.g., as part of *.refresh*). The intent here is to get the XML for the association into the clipboard.
2. *.refresh* applied to *iM* to remove the moved association. (Again, app developers if not in auto-save mode) should be sure to have invoked the *.save* method on both source and destination objects of the move, i.e., *iM* and *iM2*, before invoking the *.move* method).
3. *.\_pasteAssociation* as applied to *iM2,* i.e., to the destination itemMirror object (see item #70 in the itemMirror respository). This private method (as opposed to the proposed public method of *.pastePhantomAssociation*) will not set *localItem* to NULL. Since there should not yet be any other association (in *iM2* or the relevant xooml2.xml file) with *localItem* equal to the subfolder or file that was moved, the effect is that this pasted association will now represent the moved subfolder or file. And so all information in the association (esp. *.displayText* and the metadata of namespace elements) is also moved. Note that this private method (see issue #70) is bracketed by a *.refresh* and*.\_saveForced* no matter whether the app developer is working in auto-save mode).

With these steps, not only has the subfolder or file has been moved,, but also the associated metadata (namespace data) has been moved. And both *iM* and *iM2* should be “current” (with associations corresponding to contained subfolders and files and with values for *GUIDGeneratedOnLastWrite*  that match those of the relevant xooml2.xml files).

### What about the move of a phantom association?

The move of a phantom association is somewhat different:

1. *.deleteAssociation* is invoked on *iM* for the specified association. No matter whether the app is in auto-save mode, this method, in the context of *.move* is bracketed by *.refresh* and *.\_saveForced*. The effect is that the association is removed not only from *iM* but also from its corresponding xooml2.xml file. But the xml for the association is temporarily saved in the clipboard (see issue #70).
2. .*pastePhantomAssociation* is invoked on *iM2.* Again, no matter whether the app is in auto-save mode, when this method is used in the context of a move, it is bracketed by *.refresh* and *.\_saveForced*.

### What about other methods involving file system operations?

The methods besides *.move* that (potentially) involve file system operations are

* .*deleteAssociation* — when the association is “real” and so represents a subfolder or file.
* .*createAssociation* — in cases where a new local subfolder, file or special shortcut file are created under folder *F.* Two of these cases, 3 and 5, involve creation of a shortcut file linking, respectively, to an existing non-grouping item (e.g., a file) or an existing grouping item (e.g., a folder). Cross-platform, cross-cloud-store support for shortcuts does not appear to be imminent. So we can disregard these cases for now. Case 6, create a new, local non-grouping item (e.g., a file) likely involves extra work including UI to determine available file types and support the selection of one of these. This case will not be supported in the near future. However, **Case 7,** the creation of a new, local grouping item (e.g., a folder) is already supported and is often used in iM apps.
* .*upgradeAssociation —* converts a simple phantom association (i.e., a phantom association where associatedItem is null) into an association where *localItem* points to a newly created local subfolder (under folder *F*), i.e., where *localItem* is either the name of this subfolder (Dropbox) or the ID for this subfolder (Google Drive). The name of the subfolder can be specified as an argument (the not so aptly named,”localItemURI“, or is otherwise generated from a truncation of the association’s *displayText.*
* .*renameAssocaitionLocalItem* — involves a filesystem call through the specified itemDriver to change the name of the file or subfolder referenced in *localItem.*

For two of these commands, .*deleteAssociation* and .*createAssociation,* the work is simple.

1. Issue the file system modification to the cloud store through the specified *itemDriver.* The method fails (with appropriate error message) if the file system modification fails. Otherwise, proceed…
2. .*refresh* so that file system changes e.g. to folder *F,* are reflected in the itemMirror object *iM* and in the associated xooml2 (“xooml2.xml” under *F.*

Nothing else is required i.e., no need for .\_*saveForced.*

In the other two cases, for .*upgradeAssociation* and .*renameAssocaitionLocalItem*, things are more complicated.

1. Issue an immediate .*refresh* to insure that *iM* is in sync with associated xooml2 xml and to insure that this xooml2 is in sync with the contents of folder *F.*
2. Issue the file system modification to the cloud store through the specified *itemDriver.* The method fails (with appropriate error message) if the file system modification fails. Otherwise, proceed…
3. Do special “surgery” in *iM*so that the association involved isn’t later accidentally deleted as a consequence of *.\_sync.* This is done by setting its *localItem* to equal the name or ID of the newly created or renamed subfolder. Question: in the case of a rename of a subfolder in Google Drive, is the ID still the same? If so, then there is no need to reset *localItem.*
4. Issue a .*\_saveForced* to persist these changes of *iM* to the associated xooml2.

1. The XML might be persisted in other ways, of course, as specified through the operative xoomlDriver — for example as the text value of an attribute of a MySQL record keyed by a URI for the grouping item. [↑](#footnote-ref-1)