THE RAPTURE ENVIRONMENT

september 4, 2012

RESEARCH BY

INCAPTURE

CONTENTS

Part I.

API

1

OVERVIEW

The **Rapture** API is summarized in Figure 1. There are three general areas for the API:

1. API calls used by *user* applications.
2. API calls used to configure entities in **Rapture** . These tend to be used by configuration applications.
3. Low level API calls used to setup **Rapture** at a fundamental level.

Most API use will be at the top level through user applications.

This document goes through each API in turn, describing its general purpose and functions exposed by the API. Finally the list of *types* used in the API calls are described.

Before going through the API in detail it is worth describing how the API is used in an application or script. That is described in the next chapter.

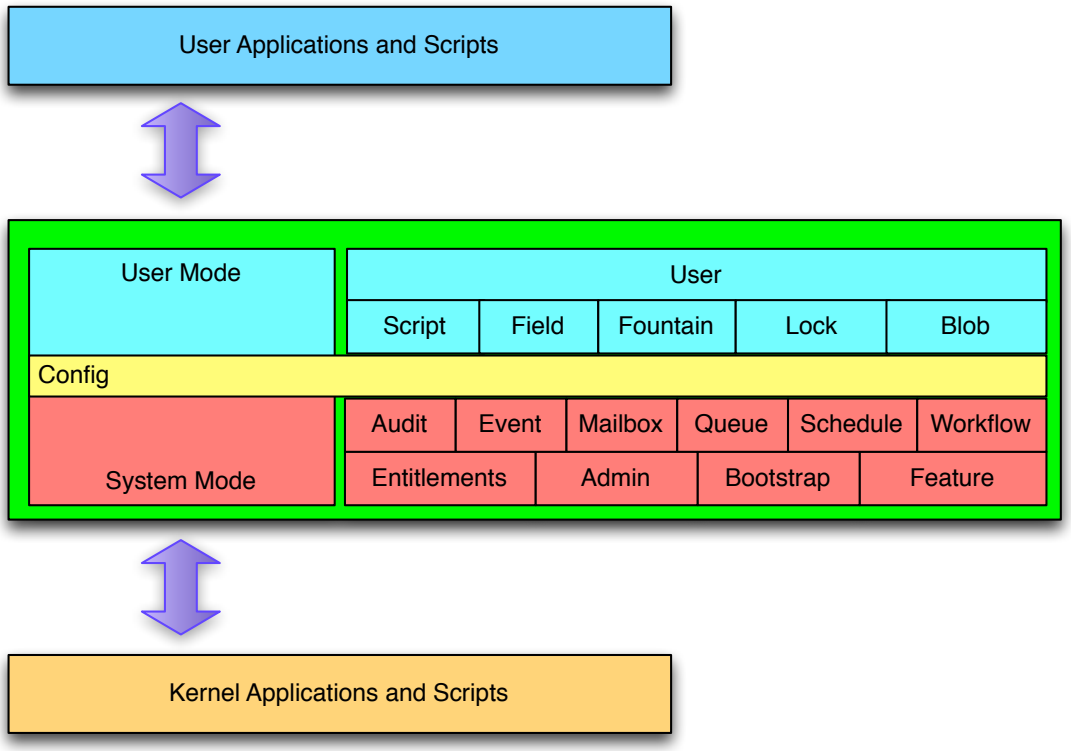

Figure 1.: The Rapture API

2

USING THE API

There are many ways to interact with **Rapture** but all interactions are through the contract defined by the **Rapture** API (described in this document). That contract is independent of the method used to connect to a **Rapture** system.

Rapture exposes the API to the *3rd* party applications in the following ways:

1. Within **Rapture** using the `Kernel` singleton object. (Used by plugins and addins of **Rapture**).
2. Within **Rapture** through the execution of **Reflex** scripts.
3. External to **Rapture** through a protocol running over `http` (usually via a load balancer to one of a set of **Rapture** endpoint servers).
   a) Through a `Python` API.
   b) Through a `Ruby` API.
   c) Through a `Go` API.
   d) Through a `Java` API.
   e) Through a `Javascript` API.
   f) Through a `Reflex` script using `ReflexRunner`.
4. Through JSP pages that run on a **Rapture** web server.
5. Through Reflex script pages that run on a **Rapture** web server.

No matter which approach is used the general flow of an API call is as follows:

1. Obtain a login API instance, connected to a **Rapture** environment.
2. Login using credentials, receive a session (a context).
3. Use that session/context in all other calls until invalidated. (Whereupon you establish credentials again).

For internally hosted access the first two steps will already have taken place within **Rapture** - the interface you use internally will already be bound to a user context taken from how the internal call is invoked or configured.

The general connectivity approach is summarized in Figure 2 on the next page.

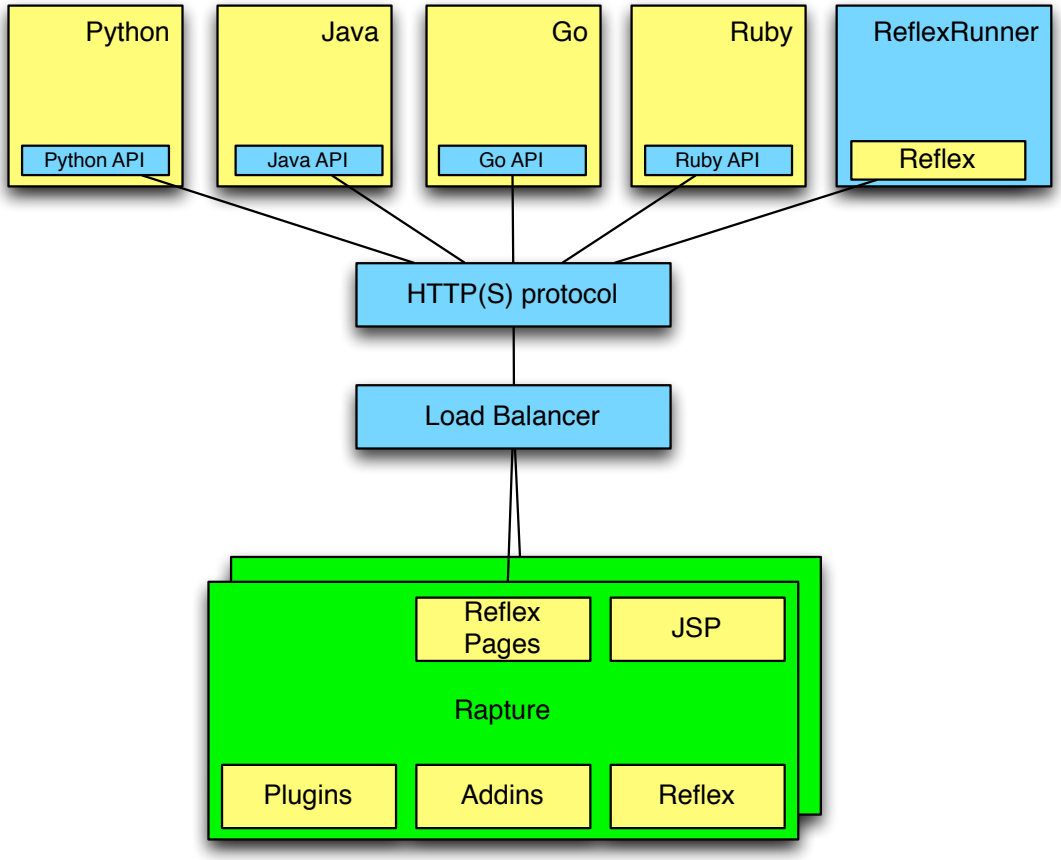

Figure 2.: Connecting to Rapture

3

ADMIN API

The Admin API is primarily used during the low level setup and initialization of a **Rapture** system. It is usually called through a user context that has administrative rights in **Rapture** (a user that has access to the entitlement path `/admin/main`). Applications such as the *FeatureInstaller* use this API and should be run in an administrative context. Great care should be taken when calling functions in this API outside of a controlled application such as *FeatureInstaller* as corruption and destruction of a **Rapture** environment can be caused through incorrect use.

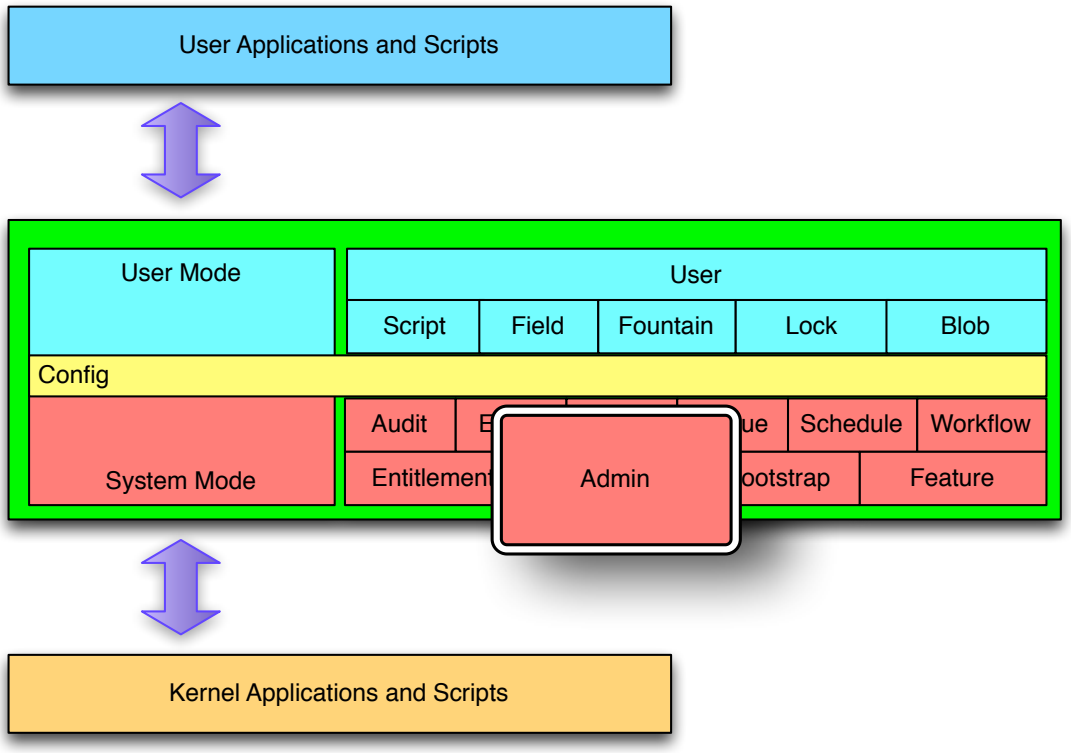

Figure 3.: Admin API

getsystemproperties

```
Map<String,String> getSystemProperties (
        List<String>    keys
)
```

```
// Reflex use
ret = #admin.getSystemProperties(keys);
```

This function retrieves the system properties in use for this instance of Rapture. As system properties are often used to control external connectivity, a client can determine the inferred connectivity endpoints by using this api call. It returns a map from system property name (the key) to value. You cannot modify the system properties of **Rapture** through the api, they are set by the administrator on startup.

---

getrepoconfig

```
List<RepoConfig> getRepoConfig (
)
```

```
// Reflex use
ret = #admin.getRepoConfig();
```

Rapture is a heirarchical set of repositories, and this method returns the configuration of the top most level - that used for general configuration and temporary (transient) values such as sessions. In clustered mode these configurations would be referencing shared storage, and in test mode they would normally refer to in-memory versions of the configuration. The type `RepoConfig`, used in this api call is defined in Part II on page 117.

---

getsessionsforuser

```
List<CallingContext> getSessionsForUser (
        String    user
)
```

```
// Reflex use
ret = #admin.getSessionsForUser(user);
```

When a user logs into **Rapture** they create a transient session, and this method is a way of retrieving all of the sessions for a given user. The `CallingContext` is a common object passed around **Rapture** api calls. It is defined in Part II on page 117.

---

getpartition

```
RapturePartition getPartition (
        String    name
)
```

```
// Reflex use
ret = #admin.getPartition(name);
```

This method is used to retrieve information about a partition in **Rapture** . Usually a call to {getPartitions}, defined on page **??**, is made to retrieve the list of partition names. A partition is a fundamental separator in **Rapture** and is used to hold types, events, scripts and the like. Entitlements can always be set at the level of a given partition, protecting all of the contents of a partition to a set of privileged users. The RapturePartition type is very basic, consisting of simply its name. It is defined in Part II on page 118.

getpartitions

```
List<RapturePartition> getPartitions (
)
```

```
// Reflex use
ret = #admin.getPartitions();
```

This method retrieves information about all of the partitions in **Rapture** , returning a list of the partition configurations. The simple RapturePartition type is documented in Part II on page 118.

createpartition

```
RapturePartition createPartition (
        String    name
)
```

```
// Reflex use
ret = #admin.createPartition(name);
```

This method creates a new partition in **Rapture** (if it does not already exist). A partition is simply defined by its unique name. The return value from this method

is the defined RapturePartition, the structure of which is documented in Part II on page 118.

doespartitionexist

```
boolean doesPartitionExist (
        String    name
)
```

```
// Reflex use
ret = #admin.doesPartitionExist(name);
```

This function provides a quick way to determine whether a partition exists in a **Rapture** system. It is often called before actually creating a partition. A typical **Reflex** script to do this is shown below:

```
    partitionName = "test";
    if !#admin.doesPartitionExist(partitionName) do
        #admin.createPartition(partitionName);
    end
```

There are distinct side effects from creating a partition that already exists – coordination a signalling activity take place within **Rapture** when changes are made and this can be minimized by not forcing the change it happen if it is not necessary.

droppartition

```
boolean dropPartition (
        String    name
        boolean    force
)
```

```
// Reflex use
ret = #admin.dropPartition(name,force);
```

Delete this partition. If the `force` flag is set the partition will be deleted even if there are still entities attached to this partition. The normal behavior is to delete the types, operations and queues associated with this partition before deleting the partition itself using this api call. You may lose data through calling this method without care and consideration.

createtype

```
RaptureType createType (
        String    partition
        String    name
        String    configuration
)
```

```
// Reflex use
ret = #admin.createType(partition,name,configuration);
```

A Rapture type is used to store data within a Rapture partition. This method creates a type, with the configuration defining how data should be stored within the type. The configuration is of the form:

```
repositoryType
   { config }
using
storageType
   { config }
```

Some examples are :

```
VREP {} using MEMORY {}
VREP {} using MONGODB { prefix = 'tms.order' }
```

The *repositoryType* can be one of *REP* - a simple key value non-versioned repository, and *VREP* - a fully auditable versioned repository.

storageType can be one of *MONGODB*, *REDIS*, *AWS*, *MEMORY*, *POSTGRES*.

The use of these storage types is defined (elsewhere).

---

doestypeexist

```
boolean doesTypeExist (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #admin.doesTypeExist(partition,name);
```

This api call can be used to determine whether a given type exists in a given partition. Creating a type is a heavyweight process in **Rapture** as all **Rapture** servers need to be informed of any change. This api call can be used to determine whether the creation step is necessary or not.

gettypesforpartition

```
List<RaptureType> getTypesForPartition (
        String    partition
)
```

```
// Reflex use
ret = #admin.getTypesForPartition(partition);
```

This method retrieves the types for a given partition. The return value is a list of RaptureType instances which are described in Part II on page 118.

gettype

```
RaptureType getType (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #admin.getType(partition,name);
```

This method returns a single type, given the partition it is in and its name. The RaptureType returned is described on Part II on page 118.

droptype

```
boolean dropType (
        String    partition
        String    name
)
```

```
// Reflex use
```

```
ret = #admin.dropType(partition,name);
```

This method removes a type and its data from the **Rapture** system. This will remove data from the system.

---

getperspectives

```
List<String> getPerspectives (
        String    partition
        String    typeName
)
```

```
// Reflex use
ret = #admin.getPerspectives(partition,typeName);
```

A type that is using a *versioned repository* will be able to use perspectives - these are different high level views of the data within that repository. For example a given repository may have an *official* perspective and a *what-if* perspective. When viewing and querying documents, the perspective allows the application designer to separate the documents into different areas that are completely separate. At a later point a perspective can be merged into another perspective.

This method retrieves the names of the perspectives associated with a given type in a partition.

---

gettags

```
List<String> getTags (
        String    partition
        String    typeName
)
```

```
// Reflex use
ret = #admin.getTags(partition,typeName);
```

A type that is using a *versioned repository* will be able to use tags - these are fixed points in time (that are invariant) that reflect what the repository looked like when the tag was created. For example a tag could be assigned to a repository that marks the *end of day* view for a repository. Any changes made after this tag was created will be accessible in the normal perspective but will not be associated with the tag.

This method returns all of the tags associated with this type in the given partition.

---

deleteuser

```
boolean deleteUser (
        String    userName
)
```

```
// Reflex use
ret = #admin.deleteUser(userName);
```

This method removes a user from this Rapture system. The user is removed from all entitlement groups also. The actual user definition is retained and marked as inactive (so the user cannot login). This is because the user may still be referenced in audit trails and the change history in type repositories.

---

adduser

```
boolean addUser (
        String    userName
        String    description
        String    hashPassword
)
```

```
// Reflex use
ret = #admin.addUser(userName,description,hashPassword);
```

This method adds a user to the Rapture environment. The user will be in no entitlement groups by default. The password field passed is actually the MD5 hash of the password - or at least the same hash function that will be applied when logging in to the system (the password is hashed, and then hashed again with the salt returned during the login protocol).

---

doesuserexist

```
boolean doesUserExist (
        String    userName
)
```

```
// Reflex use
ret = #admin.doesUserExist(userName);
```

This api call can be used to determine whether a given user exists in the **Rapture** system. Only system administrators can use this api call.

---

generateapiuser

```
RaptureUser generateApiUser (
        String    prefix
        String    description
)
```

```
// Reflex use
ret = #admin.generateApiUser(prefix,description);
```

Generates an api user, for use in connecting to Rapture in a relatively opaque way using a shared secret. An api user can log in with their access key without a password.

---

resetuserpassword

```
boolean resetUserPassword (
        String    userName
        String    newHashPassword
)
```

```
// Reflex use
ret = #admin.resetUserPassword(userName,newHashPassword);
```

This method gives an administrator the ability to reset the password of a user. The user will have the new password passed. The password parameter is actually the MD5 of the "known" password to reset - internally this will be hashed further against a unique salt for this user generated at either this point or when the user was created.

---

getremotes

```
List<RaptureRemote> getRemotes (
)
```

```
// Reflex use
ret = #admin.getRemotes();
```

Remotes are used to connect one **Rapture** cloud environment to another – the connection to a remote is done through the http api and requires an apikey user (see the method above) to be created on the remote system.

This method lists the remotes created on this system (the connections this system may make to other systems).

---

addremote

```
RaptureRemote addRemote (
        String    name
        String    description
        String    url
        String    apiKey
)
```

```
// Reflex use
ret = #admin.addRemote(name,description,url,apiKey);
```

This method adds a new remote to this system. The remote is described by its name, a description (if needed), a url to the http end point of the remote system and the name of an api user on the remote system. That remote api user must have the appropriate privileges to perform the requests necessary at the remote end.

---

removeremote

```
boolean removeRemote (
        String    name
)
```

```
// Reflex use
ret = #admin.removeRemote(name);
```

This method removes a previously created remote.

---

updateremoteapikey

```
boolean updateRemoteApiKey (
        String    name
        String    apiKey
)
```

```
// Reflex use
ret = #admin.updateRemoteApiKey(name,apiKey);
```

This method updates the user api key used by a given remote. This is an api key for the *remote* system, not this **Rapture** system.

---

setremotefortype

```
boolean setRemoteForType (
        String    partition
        String    typeName
        String    perspective
        String    remote
        String    remotePartition
        String    remoteTypeName
        String    remotePerspective
)
```

```
// Reflex use
ret = #admin.setRemoteForType(partition,typeName,perspective,remote,
    remotePartition,remoteTypeName,remotePerspective);
```

Once a remote has been defined it can be used to synchronize one type repository to another. This method defines how one (local) type is connected to a remote type. The connection is made at a perspective level, and binds the combination of a partition, typename and perspective from one system to another, using the remote already specified.

---

clearremotefortype

```
boolean clearRemoteForType (
        String    partition
        String    typeName
        String    perspective
)
```

```
// Reflex use
ret = #admin.clearRemoteForType(partition,typeName,perspective);
```

This method reverses a previously defined association between one type and a remote type.

---

pullperspective

```
boolean pullPerspective (
        String    partition
        String    typeName
        String    perspective
)
```

```
// Reflex use
ret = #admin.pullPerspective(partition,typeName,perspective);
```

If this type has a remote defined, use it to sync this repository with that of the other.

---

setoperationontype

```
boolean setOperationOnType (
        String    partition
        String    typeName
        String    opName
        String    paramDef
        String    scriptName
)
```

```
// Reflex use
ret = #admin.setOperationOnType(partition,typeName,opName,paramDef,
    scriptName);
```

An operation is effectively a script that can be run at the server, on demand by calling a named *operation* on a document within a type with a number of parameters. A typical example of an operation is one called `claimOrder` on an `order` type. When we want to claim an order we can simply call this operation by name on that document, passing in perhaps a parameter set containing the name of the trader claiming the order. Behind the scense the act of invoking an operation will call the attached script with a context set up with the document and the parameters. The goal of the script would be to update the document with the changes needed to "claim the order" and save the document back into **Rapture** . As this script execution takes place centrally within **Rapture** and not on the client site it is both more performant and more secure than executing code through the low level functional api from client side code.

This method defines an operation.

---

removeoperationfromtype

```
boolean removeOperationFromType (
        String    partition
        String    typeName
        String    opName
)
```

```
// Reflex use
ret = #admin.removeOperationFromType(partition,typeName,opName);
```

This method removes a previously added operation on a type.

---

getoperationsfortype

```
List<RaptureOperation> getOperationsForType (
        String    partition
        String    typeName
)
```

```
// Reflex use
ret = #admin.getOperationsForType(partition,typeName);
```

This method retrieves the operations that are defined for a given type.

---

getoperationfortype

```
RaptureOperation getOperationForType (
        String    partition
        String    typeName
        String    opName
)
```

```
// Reflex use
ret = #admin.getOperationForType(partition,typeName,opName);
```

This method retrieves the definition of a single operation on a type.

---

doesoperationexist

```
boolean doesOperationExist (
        String    partition
        String    typeName
        String    opName
)
```

```
// Reflex use
ret = #admin.doesOperationExist(partition,typeName,opName);
```

Does this operation exist?

---

getview

```
RaptureView getView (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #admin.getView(partition,name);
```

This method returns the configuration of a predefined view in a Rapture partition.

---

getviewsforpartition

```
List<RaptureView> getViewsForPartition (
        String    partition
)
```

```
// Reflex use
ret = #admin.getViewsForPartition(partition);
```

This method returns all of the Rapture Views associated with a partition.

---

dropview

```
boolean dropView (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #admin.dropView(partition,name);
```

This method removes a previously defined view from the system.

---

createview

```
RaptureView createView (
        String    partition
        String    name
        String    filterFn
        String    mapFn
        String    parameterSpec
        String    columnSpec
)
```

```
// Reflex use
ret = #admin.createView(partition,name,filterFn,mapFn,parameterSpec,
    columnSpec);
```

This method creates a new view in a Rapture partition. A view is defined by a set of functions (filter and map) and a definition of what format any parameters should

be passed. A final parameter defines how the results are returned. When called, a view runs the filter function against each document in the repository, and for every document that returns true, the map function will be called to generate the results that will be added to the return from the call.

---

doesviewexist

```
boolean doesViewExist (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #admin.doesViewExist(partition,name);
```

Does this view exist?

---

addtemplate

```
boolean addTemplate (
        String    name
        String    template
        boolean    overwrite
)
```

```
// Reflex use
ret = #admin.addTemplate(name,template,overwrite);
```

This function adds a template to the Rapture system. A template is a simple way of registering predefined configurations that can be used to automatically generate configurations for repositories, queues, and the like. Templates use the popular `StringTemplate` library for merging values into a text template.

---

runtemplate

```
String runTemplate (
        String    name
        String    params
)
```

```
// Reflex use
ret = #admin.runTemplate(name,params);
```

This method executes a template, replacing parts of the template with the passed parameters to create a new string.

---

gettemplate

```
String getTemplate (
        String    name
)
```

```
// Reflex use
ret = #admin.getTemplate(name);
```

This method returns the definition of a template.

---

clonetype

```
boolean cloneType (
        String    srcPartition
        String    srcType
        String    srcPerspective
        String    targPartition
        String    targType
        String    targPerspective
        boolean    wipe
)
```

```
// Reflex use
ret = #admin.cloneType(srcPartition,srcType,srcPerspective,targPartition,
   targType,targPerspective,wipe);
```

Used to (shallow) clone a types data into another type. The target type is wiped out before hand if wipe is set to true.

---

addiptowhitelist

```
boolean addIPToWhiteList (
        String    ipAddress
)
```

```
// Reflex use
ret = #admin.addIPToWhiteList(ipAddress);
```

Use this method to add an IP address to a white list of allowed IP addresses that can log in to this Rapture environment. Once set only IP addresses in this ipAddress list can access **Rapture** . By default there are no whitelist IP addresses defined which actually means that all IP addresses are allowed.

---

removeipfromwhitelist

```
boolean removeIPFromWhiteList (
        String    ipAddress
)
```

```
// Reflex use
ret = #admin.removeIPFromWhiteList(ipAddress);
```

Use this method to remove an IP address from a white list

---

getipwhitelist

```
List<String> getIPWhiteList (
)
```

```
// Reflex use
ret = #admin.getIPWhiteList();
```

Use this method to return the IP white list

---

runbatchscript

```
String runBatchScript (
```

```
            String    script
    )
```

```
// Reflex use
ret = #admin.runBatchScript(script);
```

This method runs a batch script at the target site

4

APP API

The app api is used to create and manipulate registered applications and instances of them that can be downloaded via Java Web Start.

setapplicationrepositorysettings

```
boolean setApplicationRepositorySettings (
        AppRepositorySettings    config
)
```

```
// Reflex use
ret = #app.setApplicationRepositorySettings(config);
```

Set the application repository settings for this environment. This is primarily the codebase for the environment, the type `AppRepositorySettings` is documented in Part II on page 122.

---

getapplicationrepositorysettings

```
AppRepositorySettings getApplicationRepositorySettings (
)
```

```
// Reflex use
ret = #app.getApplicationRepositorySettings();
```

This api call is used to retrieve application repository settings previously defined using the `setApplicationRepositorySettings` call.

---

getapplications

```
List<String> getApplications (
)
```

```
// Reflex use
ret = #app.getApplications();
```

This api call is used to list the applications defined in the system. Applications are defined through the `createApplicationConfig` api call, and given the list of names from this api call the `getApplicationConfig` can be used to retrieve the underlying information.

---

createapplicationconfig

```
boolean createApplicationConfig (
        String    appName
        AppConfig    config
)
```

```
// Reflex use
ret = #app.createApplicationConfig(appName,config);
```

Create an application config. An application config is used to define a potential application that can be downloaded from a **Rapture** server through Java Web Start. The application config defines the main information about an application – the location of the binaries, a description, the location of icons. Instances of applications can then be created from this config (see `createApplicationInstance` below) – it is in the application instance that an application is customized for a specific purpose through the use of command line options.

---

getapplicationconfig

```
AppConfig getApplicationConfig (
        String    appName
)
```

```
// Reflex use
ret = #app.getApplicationConfig(appName);
```

This api call is used to retrieve a previously defined application config. Application configs are created through the `createApplicationConfig` api call.

---

createapplicationinstance

```
boolean createApplicationInstance (
        String    appName
        String    instanceName
        AppInstanceConfig    instanceConfig
)
```

```
// Reflex use
ret = #app.createApplicationInstance(appName,instanceName,instanceConfig);
```

Create an application instance

---

getapplicationinstances

```
List<String> getApplicationInstances (
        String    appName
)
```

```
// Reflex use
ret = #app.getApplicationInstances(appName);
```

List application instances

---

getapplicationinstanceconfig

```
AppInstanceConfig getApplicationInstanceConfig (
        String    appName
        String    instanceName
)
```

```
// Reflex use
ret = #app.getApplicationInstanceConfig(appName,instanceName);
```

Get application instance config

---

deleteapplicationconfig

```
boolean deleteApplicationConfig (
        String    appName
)
```

```
// Reflex use
ret = #app.deleteApplicationConfig(appName);
```

Delete an application config

---

deleteapplicationinstance

```
boolean deleteApplicationInstance (
        String    appName
        String    instanceName
)
```

```
// Reflex use
ret = #app.deleteApplicationInstance(appName,instanceName);
```

Delete an application instance

5

AUDIT API

The Audit api provides a way to create special logs that are permanent records of activity in a Rapture system. Internally Rapture uses a system audit log for recording important events that take place in a Rapture environment. Users (or applications) can create their own specific audit logs for the same purpose.

The api provides a way of creating and removing these logs, and then a simple way of recording log entries. A final api call gives the caller the ability to retrieve log entries.

createauditlog

```
boolean createAuditLog (
        String    name
        String    config
)
```

```
// Reflex use
ret = #audit.createAuditLog(name,config);
```

This method creates a new audit log, given a name and a configuration string. The configuration string defines the implementation to be used to store the audit entries.

---

doesauditlogexist

```
boolean doesAuditLogExist (
        String    name
)
```

```
// Reflex use
ret = #audit.doesAuditLogExist(name);
```

Does this audit log exist

---

deleteauditlog

```
boolean deleteAuditLog (
        String    name
)
```

```
// Reflex use
ret = #audit.deleteAuditLog(name);
```

This method removes a previously created audit log.

---

getauditlog

```
AuditLogConfig getAuditLog (
        String    name
)
```

```
// Reflex use
ret = #audit.getAuditLog(name);
```

This method retrieves the configuration of a previous created audit log.

---

writeauditentry

```
boolean writeAuditEntry (
        String    name
        String    category
        int    level
        String    message
)
```

```
// Reflex use
ret = #audit.writeAuditEntry(name,category,level,message);
```

This method writes an audit entry to an audit log.

---

getrecentlogentries

```
List<AuditLogEntry> getRecentLogEntries (
        String    name
        int    count
)
```

```
// Reflex use
ret = #audit.getRecentLogEntries(name,count);
```

This method retrieves previously registered log entries, given a maximum number of entries to return.

6

BLOB API

The Blob api is used to manipulate large opaque objects that do have names (displaynames) like other data but do not have any insight to be gained from their contents from within Rapture. The RESTful API can be used to efficiently download a blob as a stream (or upload it)

createblobrepository

```
boolean createBlobRepository (
        String    partition
        String    name
        String    config
)
```

```
// Reflex use
ret = #blob.createBlobRepository(partition,name,config);
```

Creates a repository used to store blobs

---

getblobrepositoryconfig

```
RaptureBlobConfig getBlobRepositoryConfig (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #blob.getBlobRepositoryConfig(partition,name);
```

Retrieves blob repository information

---

removeblobrepository

```
boolean removeBlobRepository (
```

```
        String    partition
        String    name
        boolean    destroy
)
```

```
// Reflex use
ret = #blob.removeBlobRepository(partition,name,destroy);
```

Remove a blob repository

---

createblobfromstring

```
boolean createBlobFromString (
        String    partition
        String    displayName
        String    content
        boolean    append
)
```

```
// Reflex use
ret = #blob.createBlobFromString(partition,displayName,content,append);
```

Stores a blob in one hit, assuming a String representation. If append, adds to any content already existing

---

getblob

```
String getBlob (
        String    partition
        String    displayName
)
```

```
// Reflex use
ret = #blob.getBlob(partition,displayName);
```

Retrieves a blob in one hit, assuming a String representation

---

deleteblob

```
String deleteBlob (
        String    partition
        String    displayName
)
```

```
// Reflex use
ret = #blob.deleteBlob(partition,displayName);
```

Removes a blob from the store

---

getblobsize

```
Long getBlobSize (
        String    partition
        String    displayName
)
```

```
// Reflex use
ret = #blob.getBlobSize(partition,displayName);
```

Retrieves the size of a blob

---

getblobpart

```
String getBlobPart (
        String    partition
        String    displayName
        Long    start
        Long    size
)
```

```
// Reflex use
ret = #blob.getBlobPart(partition,displayName,start,size);
```

Retrievs part of a blob

7

BOOTSTRAP API

The Bootstrap API is used to setup an initial Rapture environment

setemphemeralrepository

```
boolean setEmphemeralRepository (
        String    config
)
```

```
// Reflex use
ret = #bootstrap.setEmphemeralRepository(config);
```

The ephemeral repository is used to store information that does not need to survive a restart of Rapture. It normally holds information such as sessions and its configuration is usually based around a shared non-versioned memory model

---

setconfigrepository

```
boolean setConfigRepository (
        String    config
)
```

```
// Reflex use
ret = #bootstrap.setConfigRepository(config);
```

The config repository is used to store general configuration information about entities in Rapture. These entities include users, types, indices, queues and the like.

---

setsettingsrepository

```
boolean setSettingsRepository (
        String    config
)
```

```
// Reflex use
ret = #bootstrap.setSettingsRepository(config);
```

The settings repository is used to store general low level settings in Rapture. (To be described further).

---

restartbootstrap

```
boolean restartBootstrap (
)
```

```
// Reflex use
ret = #bootstrap.restartBootstrap();
```

After changing the definition of any bootstrap repository, Rapture will need to be restarted. This method will restart Rapture.

---

addscriptclass

```
boolean addScriptClass (
        String    keyword
        String    className
)
```

```
// Reflex use
ret = #bootstrap.addScriptClass(keyword,className);
```

All scripts that are run by Rapture are passed a set of *helper* instances that can be used by the script. The helpers are locked to the entitlement context of the calling user. This method sets the name of such a class in this context. It is primarily an internal function, defined during startup, as the class provided must be accessible by the main Rapture application.

---

getscriptclasses

```
Map<String,String> getScriptClasses (
)
```

```
// Reflex use
ret = #bootstrap.getScriptClasses();
```

This method retrieves previous defined script classes for this system

---

removescriptclass

```
boolean removeScriptClass (
        String    keyword
)
```

```
// Reflex use
ret = #bootstrap.removeScriptClass(keyword);
```

This method removes a previously defined script class.

8

COMMIT API

Certain types of repository in Rapture are *versioned repositories*. Versioned repositories maintain a complete history of all changes that have taken place in repository. Each change is associated with a *commit* and that commit is associated with a user, has a comment, and is fixed in a point in time.

The commit api is used to retrieve information about commits that have taken place in a repository. All of the methods are read-only requests for information.

getcommithistory

```
List<RaptureCommit> getCommitHistory (
        String    partition
        String    typeName
        String    perspective
)
```

```
// Reflex use
ret = #commit.getCommitHistory(partition,typeName,perspective);
```

This method retrieves the complete commit history for this type, within a partition and locked to a perspective. (To be changed to a cursor approach).

---

getcommitssince

```
List<RaptureCommit> getCommitsSince (
        String    partition
        String    typeName
        String    perspective
        String    commitReference
)
```

```
// Reflex use
ret = #commit.getCommitsSince(partition,typeName,perspective,
    commitReference);
```

This method is primarily used by the *remote* synchronization procedure. Once end of the remote asks the other (remote) environment for all of the commits that have

taken place since since a given reference. As each commit is unique and ordered in time for a given perspective this will return all of the changes that need to be synchronized from the remote partition.

---

getdocumentobject

```
DocumentObject getDocumentObject (
        String    partition
        String    typeName
        String    reference
)
```

```
// Reflex use
ret = #commit.getDocumentObject(partition,typeName,reference);
```

This method retrieves the content of a commit – in particular the content of a document change. The method is primarily (and only) used during the remote synchronization protocol.

---

gettreeobject

```
TreeObject getTreeObject (
        String    partition
        String    typeName
        String    reference
)
```

```
// Reflex use
ret = #commit.getTreeObject(partition,typeName,reference);
```

This method retrieves the content of a commit – in particular the content of a tree of changes. The method is primarily (and only) used during the remote synchronization protocol.

---

getcommitobject

```
CommitObject getCommitObject (
        String    partition
        String    typeName
```

```
            String    reference
    )
```

```
// Reflex use
ret = #commit.getCommitObject(partition,typeName,reference);
```

Finally this method returns information about the commit itself, used by the remote protocol.

9

ENTITLEMENT API

Entitlements are a very important part of the security of **Rapture** , and the Entitlement api is the way in which information about this entitlements is updated. The api is of course protected by the same entitlements system, so care must be taken to not remove your own entitlement to this api through the *use* of this api.

Entitlements work like this. Users can be members of entitlement groups, and entitlement groups are members of entitlements. Each api call within Rapture is associated with an entitlement path, and when a user wishes to execute that api call they are checked to see if they are a member of that entitlement (by seeing which groups they are members of). Some api calls have dynamic entitlements, where the full name of the entitlement is derived from fundamental concepts such as typename, displayname, queuename etc. If an entitlement with the specific name exists that is used, otherwise the full entitlement path is truncated one part at a time until an entitlement is found.

getentitlements

```
List<RaptureEntitlement> getEntitlements (
)
```

```
// Reflex use
ret = #entitlement.getEntitlements();
```

This method is used to retrieve all of the entitlements defined in **Rapture** .

---

getentitlementgroups

```
List<RaptureEntitlementGroup> getEntitlementGroups (
)
```

```
// Reflex use
ret = #entitlement.getEntitlementGroups();
```

This method returns all of the entitlement groups defined in the **Rapture** environment.

---

addentitlement

```
RaptureEntitlement addEntitlement (
        String    name
        String    initialGroup
)
```

```
// Reflex use
ret = #entitlement.addEntitlement(name,initialGroup);
```

This method adds a new entitlement, specifying an initial group that should be assigned to this entitlement. The reason for assigning an initial group is to prevent lock out.

---

addgrouptoentitlement

```
RaptureEntitlement addGroupToEntitlement (
        String    name
        String    group
)
```

```
// Reflex use
ret = #entitlement.addGroupToEntitlement(name,group);
```

This method is used to add an entitlement group to an entitlement.

---

removegroupfromentitlement

```
RaptureEntitlement removeGroupFromEntitlement (
        String    name
        String    group
)
```

```
// Reflex use
ret = #entitlement.removeGroupFromEntitlement(name,group);
```

This method reverses the act of adding a group to an entitlement.

---

deleteentitlement

```
boolean deleteEntitlement (
        String    name
)
```

```
// Reflex use
ret = #entitlement.deleteEntitlement(name);
```

This method removes an entitlement entirely from the system.

---

deleteentitlementgroup

```
boolean deleteEntitlementGroup (
        String    name
)
```

```
// Reflex use
ret = #entitlement.deleteEntitlementGroup(name);
```

This method removes an entitlement group from the system.

---

addentitlementgroup

```
RaptureEntitlementGroup addEntitlementGroup (
        String    name
)
```

```
// Reflex use
ret = #entitlement.addEntitlementGroup(name);
```

This method adds a new entitlement group to the system.

---

addusertoentitlementgroup

```
RaptureEntitlementGroup addUserToEntitlementGroup (
        String    name
        String    user
```

```
)
```

```
// Reflex use
ret = #entitlement.addUserToEntitlementGroup(name,user);
```

This method adds a user to an existing entitlement group. The user will then have all of the privileges (entitlements) associated with that group.

---

removeuserfromentitlementgroup

```
RaptureEntitlementGroup removeUserFromEntitlementGroup (
        String    name
        String    user
)
```

```
// Reflex use
ret = #entitlement.removeUserFromEntitlementGroup(name,user);
```

This method reverses the act of the adding a user to a group.

10

EVENT API

Events are used to coordinate large scale activity in Rapture. The process is relatively simple - a caller assigns any number of scripts to a named *event* (simply a unique path), and then when the event is fired all attached scripts are scheduled for execution. Some events are internally managed (system events) and other events can be user created and managed.

attachscripttoevent

```
boolean attachScriptToEvent (
        String    partition
        String    eventName
        String    scriptName
        boolean    performOnce
)
```

```
// Reflex use
ret = #event.attachScriptToEvent(partition,eventName,scriptName,
    performOnce);
```

This method is used to attach a script to an event. A final parameter signals whether this script should be detached from the event when it is fired.

---

getevent

```
RaptureEvent getEvent (
        String    partition
        String    eventName
)
```

```
// Reflex use
ret = #event.getEvent(partition,eventName);
```

This method is used to retrieve information about an event (primarily the scripts attached to it).

---

removescriptfromevent

```
boolean removeScriptFromEvent (
        String    partition
        String    eventName
        String    scriptName
)
```

```
// Reflex use
ret = #event.removeScriptFromEvent(partition,eventName,scriptName);
```

This method detaches a script from the event.

---

fireevent

```
boolean fireEvent (
        String    partition
        String    eventName
        String    displayName
        String    eventContext
)
```

```
// Reflex use
ret = #event.fireEvent(partition,eventName,displayName,eventContext);
```

This method fires an event, scheduling any attached scripts to run. The optional displayName and context parameters are passed to the script when fired.

---

removeevent

```
boolean removeEvent (
        String    partition
        String    eventName
)
```

```
// Reflex use
ret = #event.removeEvent(partition,eventName);
```

This method removes an event (and any attached scripts) from the system. If the event is fired at a later point nothing will happen as there would be no scripts attached.

11

FEATURE API

The feature api is used to manipulate information about stored features in the system

getinstalledfeatures

```
List<FeatureConfig> getInstalledFeatures (
)
```

```
// Reflex use
ret = #feature.getInstalledFeatures();
```

List feature in the system

---

getfeature

```
FeatureConfig getFeature (
        String    name
)
```

```
// Reflex use
ret = #feature.getFeature(name);
```

Retrieve a feature by name

---

recordfeature

```
boolean recordFeature (
        FeatureConfig    feature
)
```

```
// Reflex use
ret = #feature.recordFeature(feature);
```

Record a feature installed

doesfeatureneedtobeinstalled

```
boolean doesFeatureNeedToBeInstalled (
        FeatureConfig    feature
)
```

```
// Reflex use
ret = #feature.doesFeatureNeedToBeInstalled(feature);
```

Should a feature be installed

12

FIELDS API

Fields are well known concepts in Rapture that are parts of documents

getfields

```
List<RaptureField> getFields (
        String    partition
)
```

```
// Reflex use
ret = #fields.getFields(partition);
```

---

getfieldslist

```
List<RaptureField> getFieldsList (
        String    partition
        List<String>    names
)
```

```
// Reflex use
ret = #fields.getFieldsList(partition,names);
```

---

createfield

```
RaptureField createField (
        String    partition
        String    name
        String    category
        String    description
)
```

```
// Reflex use
ret = #fields.createField(partition,name,category,description);
```

doesfieldexist

```
boolean doesFieldExist (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #fields.doesFieldExist(partition,name);
```

updatefield

```
RaptureField updateField (
        String    partition
        RaptureField    field
)
```

```
// Reflex use
ret = #fields.updateField(partition,field);
```

deletefield

```
boolean deleteField (
        String    partition
        String    name
        String    category
)
```

```
// Reflex use
ret = #fields.deleteField(partition,name,category);
```

retrievefieldsfromdocument

```
List<String> retrieveFieldsFromDocument (
```

```
        String    partition
        String    perspective
        String    displayName
        List<String>    fields
    )
```

```
// Reflex use
ret = #fields.retrieveFieldsFromDocument(partition,perspective,displayName
    ,fields);
```

retrievefieldsfromcontent

```
    List<String> retrieveFieldsFromContent (
        String    partition
        String    perspective
        String    displayName
        String    content
        List<String>    fields
    )
```

```
// Reflex use
ret = #fields.retrieveFieldsFromContent(partition,perspective,displayName,
    content,fields);
```

13

FOUNTAIN API

A fountain is a unique number generator - once defined it can be used to create unique ids that can be attached to documents or entities. Fountains can be attached to a type so that new documents created for that type can optionally have unique ids.

getfountains

```
List<RaptureFountainConfig> getFountains (
)
```

```
// Reflex use
ret = #fountain.getFountains();
```

This method is used to retrieve information about the defined fountains in the Rapture system.

---

createfountain

```
RaptureFountainConfig createFountain (
        String    partition
        String    name
        String    config
)
```

```
// Reflex use
ret = #fountain.createFountain(partition,name,config);
```

This method is used to define a new fountain in a given partition. The configuration parameter defines the storage to be used for managing the fountain.

---

doesfountainexist

```
boolean doesFountainExist (
```

```
        String    partition
        String    name
)
```

```
// Reflex use
ret = #fountain.doesFountainExist(partition,name);
```

Does this fountain exist

---

deletefountain

```
boolean deleteFountain (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #fountain.deleteFountain(partition,name);
```

This method is used to delete a previously defined fountain.

---

resetfountain

```
boolean resetFountain (
        String    partition
        String    name
        Long    count
)
```

```
// Reflex use
ret = #fountain.resetFountain(partition,name,count);
```

This method can be used to reset a fountain to a new id - all future requests will start from this new point.

---

incrementfountain

```
String incrementFountain (
```

```
        String    partition
        String    name
        Long    amount
)
```

```
// Reflex use
ret = #fountain.incrementFountain(partition,name,amount);
```

This method is used to increment the fountain and return a string that corresponds to the newly generated id.

---

addfountaintotype

```
RaptureType addFountainToType (
        String    partition
        String    name
        String    typeName
)
```

```
// Reflex use
ret = #fountain.addFountainToType(partition,name,typeName);
```

This method associates a fountain with a type, so that when a document containing an autoid string is created that autoid will be replaced with a unique id.

---

getlockprovidersforpartition

```
List<RaptureLockConfig> getLockProvidersForPartition (
        String    partition
)
```

```
// Reflex use
ret = #lock.getLockProvidersForPartition(partition);
```

---

createlockprovider

```
RaptureLockConfig createLockProvider (
        String    partition
        String    lockName
        String    config
        String    pathPosition
)
```

```
// Reflex use
ret = #lock.createLockProvider(partition,lockName,config,pathPosition);
```

---

doeslockproviderexist

```
boolean doesLockProviderExist (
        String    partition
        String    lockName
)
```

```
// Reflex use
ret = #lock.doesLockProviderExist(partition,lockName);
```

---

getlockprovider

```
RaptureLockConfig getLockProvider (
        String    partition
        String    lockName
)
```

```
// Reflex use
ret = #lock.getLockProvider(partition,lockName);
```

deletelockprovider

```
boolean deleteLockProvider (
        String    partition
        String    lockName
)
```

```
// Reflex use
ret = #lock.deleteLockProvider(partition,lockName);
```

acquirelock

```
boolean acquireLock (
        String    partition
        String    lockProvider
        String    lockName
        long    secondsToWait
        long    secondsToKeep
)
```

```
// Reflex use
ret = #lock.acquireLock(partition,lockProvider,lockName,secondsToWait,
   secondsToKeep);
```

releaselock

```
boolean releaseLock (
```

```
            String    partition
            String    lockProvider
            String    lockName
    )
```

```
// Reflex use
ret = #lock.releaseLock(partition,lockProvider,lockName);
```

15

MAILBOX API

Each Rapture environment has a single mailbox, although it is divided into logical parts by partition and category. Remote users (usually remote systems) can submit items to a mailbox, whereupon an event is signalled to allow for any processing of that item. Typical processing validates the mailbox content and creates real entities within the local system (e.g. an incoming order is convertered into a real order if valid). When processed the category of an item can be changed to 'Done' to ensure it isn't reprocessed

postmailboxmessage

```
String postMailboxMessage (
        String    partition
        String    category
        String    content
)
```

```
// Reflex use
ret = #mailbox.postMailboxMessage(partition,category,content);
```

This method is used to post a message onto a category (for a partition).

---

movemailboxmessage

```
boolean moveMailboxMessage (
        String    partition
        String    oldCategory
        String    id
        String    category
)
```

```
// Reflex use
ret = #mailbox.moveMailboxMessage(partition,oldCategory,id,category);
```

Move a message to another category

---

setmailboxstorage

```
boolean setMailboxStorage (
        String    mailboxConfig
        String    fountainConfig
)
```

```
// Reflex use
ret = #mailbox.setMailboxStorage(mailboxConfig,fountainConfig);
```

Define the configuration for mailbox storage

---

getmailboxmessages

```
List<RaptureMailMessage> getMailboxMessages (
        String    partition
        String    category
)
```

```
// Reflex use
ret = #mailbox.getMailboxMessages(partition,category);
```

Retrieve mailbox messages for a category

16

QUEUE API

The **Rapture** Queue api is used to manipulate queue entities in the system. A queue is associated with a partition and has a name. The configuration of a queue defines what the underlying implementation is – currently a queue can be created on MongoDB, Redis or, for testing only, in memory. Any number of external processes can listen for messages on a queue by calling the `getItemFromQueue` api call. If successful a `QueueTask` structure will be returned. If there are no messages on the queue a null value will be returned.

A queue guarantees that only one application will receive a given message (or `QueueTask`). However an application must inform the sub-system that the item has been worked on by calling the `markQueueItem` api call. If the message is received by an application but not marked as *done* the queueing sub-system will eventually return the task to the queue to be picked off by another application. This is to handle remote application failures.

getqueuesforpartition

```
List<RaptureQueueConfig> getQueuesForPartition (
        String    partition
)
```

```
// Reflex use
ret = #queue.getQueuesForPartition(partition);
```

---

createqueue

```
RaptureQueueConfig createQueue (
        String    partition
        String    queueName
        String    config
)
```

```
// Reflex use
ret = #queue.createQueue(partition,queueName,config);
```

---

getqueue

```
RaptureQueueConfig getQueue (
        String    partition
        String    queueName
)
```

```
// Reflex use
ret = #queue.getQueue(partition,queueName);
```

deletequeue

```
boolean deleteQueue (
        String    partition
        String    queueName
)
```

```
// Reflex use
ret = #queue.deleteQueue(partition,queueName);
```

putitemonqueue

```
String putItemOnQueue (
        String    partition
        String    queueName
        String    content
)
```

```
// Reflex use
ret = #queue.putItemOnQueue(partition,queueName,content);
```

getitemfromqueue

```
QueueTask getItemFromQueue (
        String    partition
        String    queueName
```

```
        int    waitInSeconds
    )
```

```
// Reflex use
ret = #queue.getItemFromQueue(partition,queueName,waitInSeconds);
```

markqueueitem

```
    boolean markQueueItem (
        String    partition
        String    queueName
        String    id
        boolean    done
    )
```

```
// Reflex use
ret = #queue.markQueueItem(partition,queueName,id,done);
```

createsimplejob

```
boolean createSimpleJob (
        String    group
        String    name
        String    partition
        String    script
        String    interval
        int    every
        int    maxCount
        String    jobParams
)
```

```
// Reflex use
ret = #schedule.createSimpleJob(group,name,partition,script,interval,every
    ,maxCount,jobParams);
```

In this job, interval can be "HOUR", "DAY", "SECOND", "MINUTE", .. and every is how many times (every 3 seconds etc.) The maxCount is the number of times this should be executed. 0 means "forever"

---

createcronjob

```
boolean createCronJob (
        String    group
        String    name
        String    partition
        String    script
        String    cronExpression
        String    jobParams
)
```

```
// Reflex use
ret = #schedule.createCronJob(group,name,partition,script,cronExpression,
    jobParams);
```

---

createcalendarjob

```
boolean createCalendarJob (
        String    group
        String    name
        String    partition
        String    script
        String    interval
        int    count
        String    jobParams
)
```

```
// Reflex use
ret = #schedule.createCalendarJob(group,name,partition,script,interval,
    count,jobParams);
```

createdailyintervaljob

```
boolean createDailyIntervalJob (
        String    group
        String    name
        String    partition
        String    script
        String    daysOfWeek
        String    startTimeOfDay
        String    endTimeOfDay
        int    repeatCount
        int    repeatInterval
        String    intervalType
        String    jobParams
)
```

```
// Reflex use
ret = #schedule.createDailyIntervalJob(group,name,partition,script,
    daysOfWeek,startTimeOfDay,endTimeOfDay,repeatCount,repeatInterval,
    intervalType,jobParams);
```

deletejob

```
boolean deleteJob (
```

```
            String    group
            String    name
    )
```

```
// Reflex use
ret = #schedule.deleteJob(group,name);
```

---

getjobdetails

```
    List<RaptureJobDetail> getJobDetails (
    )
```

```
// Reflex use
ret = #schedule.getJobDetails();
```

18

SCRIPT API

The Scripting API is used to define scripts that are used within Rapture

createscript

```
RaptureScript createScript (
        String    partition
        String    name
        RaptureScriptLanguage    language
        RaptureScriptPurpose    purpose
        String    script
)
```

```
// Reflex use
ret = #script.createScript(partition,name,language,purpose,script);
```

---

doesscriptexist

```
boolean doesScriptExist (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #script.doesScriptExist(partition,name);
```

---

deletescript

```
boolean deleteScript (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #script.deleteScript(partition,name);
```

getscriptnames

```
List<String> getScriptNames (
        String    partition
)
```

```
// Reflex use
ret = #script.getScriptNames(partition);
```

getscript

```
RaptureScript getScript (
        String    partition
        String    name
)
```

```
// Reflex use
ret = #script.getScript(partition,name);
```

putscript

```
RaptureScript putScript (
        String    partition
        RaptureScript    script
)
```

```
// Reflex use
ret = #script.putScript(partition,script);
```

runscript

```
String runScript (
        String    partition
        String    name
        String    paramsAsString
)
```

```
// Reflex use
ret = #script.runScript(partition,name,paramsAsString);
```

19

USER API

getwhoami

```
RaptureUser getWhoAmI (
)
```

```
// Reflex use
ret = #user.getWhoAmI();
```

updatemydescription

```
RaptureUser updateMyDescription (
        String    description
)
```

```
// Reflex use
ret = #user.updateMyDescription(description);
```

changemypassword

```
RaptureUser changeMyPassword (
        String    oldHashPassword
        String    newHashPassword
)
```

```
// Reflex use
ret = #user.changeMyPassword(oldHashPassword,newHashPassword);
```

info

```
boolean info (
)
```

```
// Reflex use
ret = #user.info();
```

getcontextinfo

```
RaptureContextInfo getContextInfo (
)
```

```
// Reflex use
ret = #user.getContextInfo();
```

setcontextpartition

```
RaptureContextInfo setContextPartition (
        String    partitionName
)
```

```
// Reflex use
ret = #user.setContextPartition(partitionName);
```

setcontextperspective

```
RaptureContextInfo setContextPerspective (
        String    perspective
)
```

```
// Reflex use
ret = #user.setContextPerspective(perspective);
```

getcontent

```
String getContent (
        String    displayName
)
```

```
// Reflex use
ret = #user.getContent(displayName);
```

---

getcontentp

```
String getContentP (
        String    partitionName
        String    perspective
        String    displayName
)
```

```
// Reflex use
ret = #user.getContentP(partitionName,perspective,displayName);
```

---

batchget

```
List<String> batchGet (
        String    partitionName
        String    perspective
        List<String>    displayNames
)
```

```
// Reflex use
ret = #user.batchGet(partitionName,perspective,displayNames);
```

Returns a list of contents (null or zero length for those that do not exist) given a list of display names

---

batchexist

```
List<boolean> batchExist (
```

```
        String    partitionName
        String    perspective
        List<String>    displayNames
)
```

```
// Reflex use
ret = #user.batchExist(partitionName,perspective,displayNames);
```

Returns a list of true/false statements on whether displaynames in the given vector exist in the system

---

putcontent

```
String putContent (
        String    displayName
        String    comment
        String    content
)
```

```
// Reflex use
ret = #user.putContent(displayName,comment,content);
```

---

putcontentp

```
String putContentP (
        String    partition
        String    perspective
        String    displayName
        String    comment
        String    content
)
```

```
// Reflex use
ret = #user.putContentP(partition,perspective,displayName,comment,content)
    ;
```

---

putcontentv

```
List<String> putContentV (
        String    partition
        String    perspective
        List<String>    displayNames
        String    comment
        String    content
)
```

```
// Reflex use
ret = #user.putContentV(partition,perspective,displayNames,comment,content
    );
```

---

deletecontent

```
boolean deleteContent (
        String    displayName
        String    comment
)
```

```
// Reflex use
ret = #user.deleteContent(displayName,comment);
```

---

deletecontentp

```
boolean deleteContentP (
        String    partition
        String    perspective
        String    displayName
        String    comment
)
```

```
// Reflex use
ret = #user.deleteContentP(partition,perspective,displayName,comment);
```

---

createtag

```
boolean createTag (
        String    typeName
        String    tagName
)
```

```
// Reflex use
ret = #user.createTag(typeName,tagName);
```

gettagcontent

```
String getTagContent (
        String    tagName
        String    displayName
)
```

```
// Reflex use
ret = #user.getTagContent(tagName,displayName);
```

deletetag

```
boolean deleteTag (
        String    typeName
        String    tagName
)
```

```
// Reflex use
ret = #user.deleteTag(typeName,tagName);
```

getcommentary

```
List<CommentaryObject> getCommentary (
        String    displayName
)
```

```
// Reflex use
ret = #user.getCommentary(displayName);
```

Retrieve the commentary associated with this displayName (which can be a path)

---

addcommentary

```
boolean addCommentary (
        String    displayName
        String    commentKey
        String    description
        String    ref
)
```

```
// Reflex use
ret = #user.addCommentary(displayName,commentKey,description,ref);
```

Add a comment to this path in the repository, in the current partition, perspective

---

folderquery

```
List<String> folderQuery (
        String    partition
        String    perspective
        String    displayNamePart
        int    depth
)
```

```
// Reflex use
ret = #user.folderQuery(partition,perspective,displayNamePart,depth);
```

---

runview

```
RaptureViewResult runView (
        String    viewName
        String    viewContext
        String    params
```

```
)
```

```
// Reflex use
ret = #user.runView(viewName,viewContext,params);
```

---

runtextsearch

```
List<RaptureSearchResult> runTextSearch (
        String    indexName
        String    searchText
        int    maxCount
)
```

```
// Reflex use
ret = #user.runTextSearch(indexName,searchText,maxCount);
```

---

runoperation

```
String runOperation (
        String    typeName
        String    operation
        String    ctx
        String    params
)
```

```
// Reflex use
ret = #user.runOperation(typeName,operation,ctx,params);
```

---

runfiltercubeview

```
RaptureCubeResult runFilterCubeView (
        String    partition
        String    typeName
        String    perspective
        String    filterFn
        String    filterParams
```

```
        String    groupFields
        String    columnFields
    )
```

```
// Reflex use
ret = #user.runFilterCubeView(partition,typeName,perspective,filterFn,
    filterParams,groupFields,columnFields);
```

---

createdncursor

```
    RaptureDNCursor createDNCursor (
        String    partition
        String    typeName
        int    count
    )
```

```
// Reflex use
ret = #user.createDNCursor(partition,typeName,count);
```

---

getnextdncursor

```
    RaptureDNCursor getNextDNCursor (
        RaptureDNCursor    cursor
        int    count
    )
```

```
// Reflex use
ret = #user.getNextDNCursor(cursor,count);
```

---

runnativequery

```
    RaptureQueryResult runNativeQuery (
        String    partition
        String    typeName
        String    repoType
        List<String>    queryParams
    )
```

```
// Reflex use
ret = #user.runNativeQuery(partition,typeName,repoType,queryParams);
```

runnativefiltercubeview

```
RaptureCubeResult runNativeFilterCubeView (
        String    partition
        String    typeName
        String    repoType
        List<String>    queryParams
        String    groupFields
        String    columnFields
)
```

```
// Reflex use
ret = #user.runNativeFilterCubeView(partition,typeName,repoType,
   queryParams,groupFields,columnFields);
```

Very much like a filter cube view, except that the initial content is taken from a native query executed against a REP repository instead of running through the view sub system.

20

WORKFLOW API

The workflow api is used to manipulate workflows - series of steps that are executed in turn as part of a run

createworkflow

```
boolean createWorkflow (
        String    name
        WorkflowConfig    config
)
```

```
// Reflex use
ret = #workflow.createWorkflow(name,config);
```

Create a workflow

---

deleteworkflow

```
boolean deleteWorkflow (
        String    name
)
```

```
// Reflex use
ret = #workflow.deleteWorkflow(name);
```

Delete a workflow

---

getworkflows

```
List<String> getWorkflows (
)
```

```
// Reflex use
ret = #workflow.getWorkflows();
```

List workflows

updateworkflow

```
boolean updateWorkflow (
        String    name
        WorkflowConfig    config
)
```

```
// Reflex use
ret = #workflow.updateWorkflow(name,config);
```

Update a workflow

getworkflow

```
WorkflowConfig getWorkflow (
        String    name
)
```

```
// Reflex use
ret = #workflow.getWorkflow(name);
```

Get a workflow

prepareworkflow

```
String prepareWorkflow (
        String    name
        Map<String,String>    params
)
```

```
// Reflex use
ret = #workflow.prepareWorkflow(name,params);
```

Prepare a workflow

runworkflow

```
boolean runWorkflow (
        String    workflowId
        String    statusId
)
```

```
// Reflex use
ret = #workflow.runWorkflow(workflowId,statusId);
```

Run a workflow

---

cancelworkflow

```
boolean cancelWorkflow (
        String    workflowId
        String    statusId
)
```

```
// Reflex use
ret = #workflow.cancelWorkflow(workflowId,statusId);
```

Cancel a running workflow

---

getworkflowstatus

```
WorkflowStatus getWorkflowStatus (
        String    workflowId
        String    statusId
)
```

```
// Reflex use
ret = #workflow.getWorkflowStatus(workflowId,statusId);
```

Get the status of a workflow

---

bumpworkflow

```
WorkflowStatus bumpWorkflow (
        String    workflowId
        String    statusId
)
```

```
// Reflex use
ret = #workflow.bumpWorkflow(workflowId,statusId);
```

Bump a workflow forward one step

---

removeworkflowstatus

```
boolean removeWorkflowStatus (
        String    workflowId
        String    statusId
)
```

```
// Reflex use
ret = #workflow.removeWorkflowStatus(workflowId,statusId);
```

Remove the status for a workflow

---

getworkflowruns

```
List<String> getWorkflowRuns (
        String    name
)
```

```
// Reflex use
ret = #workflow.getWorkflowRuns(name);
```

Get the runs for a workflow

---

getrunningworkflow

```
List<String> getRunningWorkflow (
```

)

```
// Reflex use
ret = #workflow.getRunningWorkflow();
```

Get the running workflows

Part II.

Types

rapturequeryresult

A return value from a native query

```
type RaptureQueryResult {
}
```

---

workflowconfig

This is a workflow configuration

```
type WorkflowConfig {
}
```

---

workflowstatus

This is the status of a running workflow

```
type WorkflowStatus {
}
```

---

rapturecuberesult

This is the result of a FilterCubeView

```
type RaptureCubeResult {
}
```

---

rapturedncursor

This is a displayNameQuery cursor

```
type RaptureDNCursor {
}
```

---

raptureblobconfig

This is the configuration of a BlobRepository

```
type RaptureBlobConfig {
       String   name
       String   partition
       String   config
}
```

rapturemailmessage

This is a mailbox message, usually posted by an external user

```
type RaptureMailMessage {
       String   id
       String   partition
       String   category
       String   content
       Date   when
       String   who
}
```

rapturefield

A RaptureField is the definition of a concept in Rapture, referenced within a type (or a series of types)

```
type RaptureField {
       String   partition
       String   category
       String   name
       String   longName
       String   description
       String   units
}
```

commentaryobject

This is the commentary object documentation

```
type CommentaryObject {
       String   message
       Date   when
       String   commentaryKey
       String   ref
       String   who
}
```

---

auditlogentry

```
type AuditLogEntry {
       String   category
       Date   when
       String   message
       String   user
       String   logId
}
```

---

auditlogconfig

```
type AuditLogConfig {
       String   name
       String   config
}
```

---

commitobject

```
type CommitObject {
       String   treeRef
       String   user
       Date   when
       String   comment
```

```
        String   changes
        List(String)   docReferences
        List(String)   treeReferences
}
```

documentobject

```
type DocumentObject {
        String   content
}
```

treeobject

```
type TreeObject {
        Map(String,String)   trees
        List(DocumentBagReference)   documents
}
```

raptureremote

```
type RaptureRemote {
        String   name
        String   description
        String   url
        String   apiKey
}
```

raptureprocessinstance

```
type RaptureProcessInstance {
        String   processId
        String   processGroupName
        String   instanceName
```

```
        Date   lastSeen
        RaptureProcessState   state
        long   totalTasksServiced
        float   serviceRate
        float   recentRate
}
```

---

raptureprocessgroup

```
type RaptureProcessGroup {
        String   name
        Boolean   autoAssign
        Map(String,String)   capabilities
        Map(String,List(Queues))   queuesPerPartition
}
```

---

rapturescript

```
type RaptureScript {
        String   name
        String   script
        RaptureScriptLanguage   language
        RaptureScriptPurpose   purpose
        String   partition
}
```

---

rapturescriptlanguage

```
type RaptureScriptLanguage {
        String   RUBY
        String   JAVASCRIPT
        String   PYTHON
}
```

---

rapturescriptpurpose

```
type RaptureScriptPurpose {
       String   INDEXGENERATOR
       String   MAP
       String   FILTER
       String   OPERATION
       String   PROGRAM
}
```

rapturelockconfig

```
type RaptureLockConfig {
       String   name
       String   config
       String   partition
       String   pathPosition
}
```

rapturequeueconfig

```
type RaptureQueueConfig {
       String   name
       String   config
       String   partition
}
```

rapturefountainconfig

```
type RaptureFountainConfig {
       String   name
       String   config
       String   partition
}
```

raptureindexconfig

```
type RaptureIndexConfig {
       String   name
       String   config
       String   partition
}
```

---

rapturesearchresult

```
type RaptureSearchResult {
       String   perspective
       String   displayName
}
```

---

rapturefulltextindexconfig

```
type RaptureFullTextIndexConfig {
       String   name
       String   config
       String   partition
}
```

---

repoconfig

```
type RepoConfig {
       String   name
       String   configuration
}
```

---

callingcontext

```
type CallingContext {
```

```
        String   user
        String   context
        String   salt
        Boolean   valid
}
```

rapturecontextinfo

```
type RaptureContextInfo {
        String   sessionId
        String   partition
        String   perspective
}
```

rapturepartition

```
type RapturePartition {
        String   name
}
```

rapturetype

```
type RaptureType {
        String   typeName
        String   description
        String   partitionName
        String   repositoryConfig
        String   fountain
        List(IndexScriptPair)   indexes
        List(FullTextIndexScriptPair)   fullTextIndexes
        List(RaptureOperation)   operations
        String   updateQueue
}
```

rapturecommit

```
type RaptureCommit {
       String   who
       Date   when
       String   comment
       String   changes
       String   reference
       List(String)   docReferences
       List(String)   treeReferences
}
```

---

raptureuser

```
type RaptureUser {
       String   username
       String   hashPassword
       String   description
       Boolean   inactive
       Boolean   apiKey
}
```

---

raptureview

```
type RaptureView {
       String   partitionName
       String   viewName
       String   mapFn
       String   filterFn
       List(String)   parameterNames
       List(String)   columnNames
}
```

---

raptureviewresult

```
type RaptureViewResult {
```

```
        List(String)   columnNames
        List(Object)   rows
        List(Objecy)   currentRow
}
```

raptureentitlement

```
type RaptureEntitlement {
        String   name
        EntitlementType   entType
        List(String)   groups
}
```

raptureentitlementgroup

```
type RaptureEntitlementGroup {
        String   name
        List(String)   users
}
```

raptureoperation

```
type RaptureOperation {
        String   opName
        String   paramDef
        String   scriptName
}
```

queuetask

```
type QueueTask {
        String   taskId
        String   groupId
```

```
        String   requiredGroupId
        TaskState   state
        String   executionContent
        String   completionContent
        String   taskContext
        String   param1
        String   param2
        String   partition
        TaskType   taskType
}
```

---

rapturejob

```
type RaptureJob {
        String   description
        String   jobClassName
        String   group
        String   name
        JobDataMap   dataMap
}
```

---

rapturetrigger

```
type RaptureTrigger {
        String   group
        String   name
        String   description
        Date   endTime
        Date   startTime
        Date   finalFireTime
        Date   nextFireTime
        String   fireInstanceId
        String   key
        int   priority
}
```

---

raptureevent

```
type RaptureEvent {
       String   partitionName
       String   eventName
       List(RaptureEventScript)   scripts
}
```

rapturejobdetail

```
type RaptureJobDetail {
       String   description
}
```

featureconfig

```
type FeatureConfig {
}
```

featureversion

```
type FeatureVersion {
}
```

apprepositorysettings

```
type AppRepositorySettings {
}
```

appconfig

```
type AppConfig {
}
```

---

appinstanceconfig

```
type AppInstanceConfig {
}
```

DISCLAIMER

INCAPTURE TECHNOLOGIES LLC
183 Madison Avenue, Suite 801
New York, NY 10016

RESEARCH BY
INCAPTURE