Rapture Platform

Architecture Overview

Document version 1.02, October 2014

Contents

Introduction

The server-based Rapture platform is designed for creating and handling distributed applications in a cloud environment.

Unlike other cloud systems, Rapture provides the power for handling very large sets of data and using them to provide actionable intelligence quickly. Rapture handles “back of the house” activity, so that clients can develop their own applications for enabling their own business decisions.

Audience

This manual is intended for developers and software architects using Rapture at client companies.

Supported Hardware and Software

This manual applies to versions 1.1.16 of the Rapture Platform and to later versions.

Rapture Environment

A Rapture environment is a collection of servers, cloud-based or otherwise, that are running programs that bind to an underlying storage and messaging system. On a server, the Rapture platform behaves as a kernel running several key subsystems, and is often referred to as the "Rapture Core."

Rapture kernel

Figure 1 illustrates the subsystems associated with an instance of Rapture.

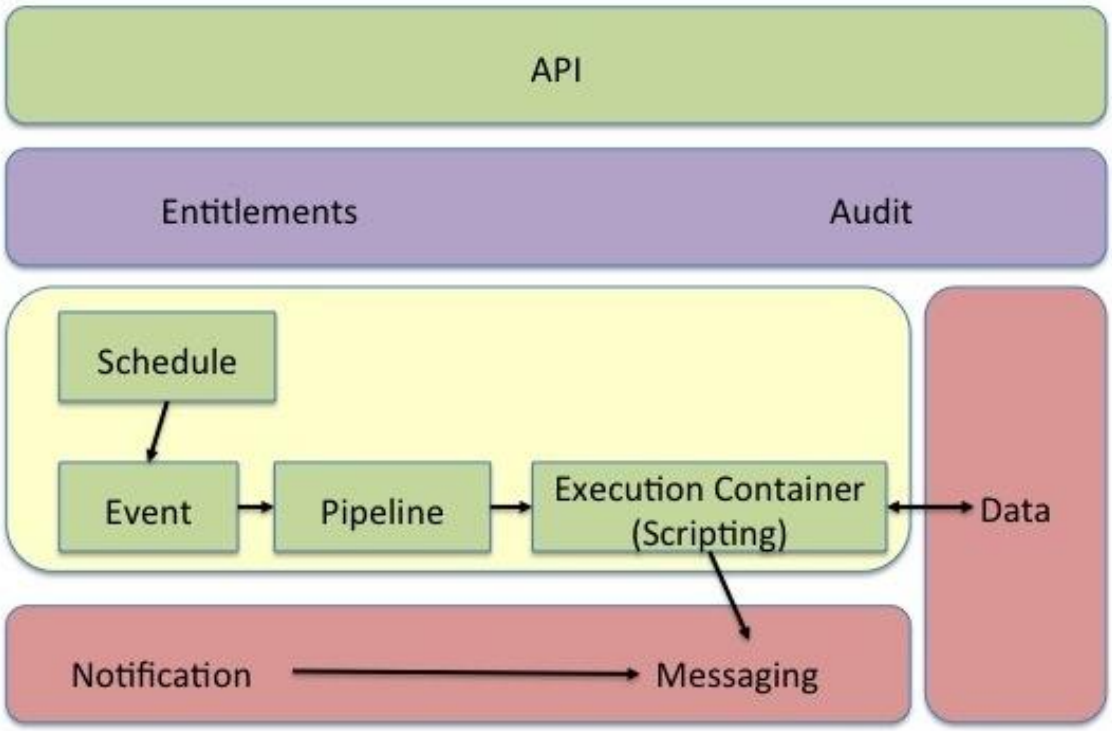

Figure 1. Rapture system overview

The Data block in this figure refers to repositories that are shielded from applications that use Rapture. These applications access the data only through the Rapture API.

Subsystems

The most fundamental part of Rapture consists of *events* and the scripts that work with them. An event typically invokes a standard or user-defined *script* when it is triggered. As shown in Figure 1, events can be *pipelined* and stored for asynchronous handling, or they can be triggered immediately. At the other end of the event chain, a trigger can be tied to a *schedule*, as shown in Figure 1, or it can be an "event hook" associated with a specific activity within Rapture, such as when a particular block of data is saved.

On a higher level, Rapture has functionality to maintain an *audit* trail for all of its activities, along with a data set for user-based privileges, called *entitlements*. A specific entitlement is associated with a group of one or more users (the “entitlement group”), and each group is entitled to make calls to a predefined list of functions within the Rapture API.

Rapture also maintains *notification* channels to enable instantaneous messaging among users, among programs, or between users and programs; for example, to alert the user to a required update.

Finally, Rapture supports a *Decision Process* feature, which allows developers to create complex workflows for multi-step processes that might require authorization from other users to proceed.

Each of these subsystems is described in more detail later in this document. Refer to the Table of Contents for their locations.

Pre-built server processes

The Rapture platform uses a variety of pre-built server processes to put the Rapture kernel into an execution mode.

Of these server processes, the most commonly used is the Rapture API Server, which provides an HTTP transport layer to client applications outside Rapture so that they are able to access the Rapture API. In addition, the Rapture API Server provides a configuration context for binding Rapture to underlying data repositories and messaging systems.

Other important pre-built server processes include the *Schedule Server*, which coordinates the event schedule entries and the invoking of referenced tasks; the *Compute Server*, which is simply an execution container for Reflex scripts; and *Rapture Runner*, which can start or stop Rapture processes with identical configurations on a given server. In particular, Rapture Runner is used for invoking the Rapture API Server, the Schedule Server, and duplicate instances of Rapture Runner, in addition to other processes.

Data

One of the key features of the Rapture platform is its ability to manage data securely. All data in Rapture is classified as either managed or presented.

Managed data is accessible only through Rapture and can never be modified by client applications. This feature gives Rapture full control over how the data is stored, whether it is versioned, and what details to include in a change log.

Presented data can be accessed and modified outside Rapture, but it is usually read-only to Rapture applications. Using presented data is extremely helpful for data migration and allows for very easy setup within Rapture. Presented data can also be useful for secondary purposes, such as supplemental reports. However, the platform features associated with managed data cannot be used.

Data classes

The Rapture platform defines four classes of data, and each of its data repositories can be associated with only one of these classes. This model helps to ensure a systematic way to reference all data through URIs, as discussed in the next section. The four data classes are: document, series, blob, and sheet.

IMPORTANT: Data types in Java and other languages are not related to data classes. A *String* type, for example, can be stored as a document, a series, or a blob.

A *document* is a piece of text that is usually formatted as a JSON structure. Because of JSON's portability and ease of use, the document class is widely used in Rapture applications. The Rapture platform also uses documents to describe the internal configuration of its own core system.

A *series* is a keyed list of data points. In most applications, the key is a timestamp, and therefore the series represents how the data changes with time. The data points within a series can be simple types, such as numbers, or they can be data structures.

When a series is stored in a Rapture repository, its keys are aligned so that data can be extracted and analyzed more efficiently, as shown in Figure 2. In this example, each horizontal row represents a different series of prices, but they are all aligned vertically with a keyed timestamp. Other sets of series could be aligned with, for instance, a column name as the key.

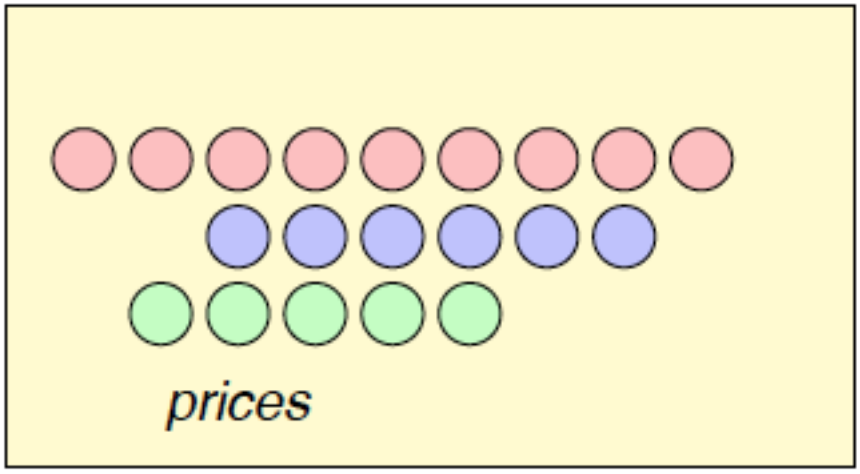

Figure 2. How price data is stored as a series

The *sheet* data class is modeled after spreadsheets. All data within a sheet is stored in a *cell* structure, with each cell having a row and column coordinate. Cells can also contain data for formatting and running calculations on the contents, so that the sheet can be presented in a user interface without requiring any format translation.

A *blob* in Rapture is simply binary data without any other classifications. It is useful for storing items such as images or PDF data.

Data repositories

Each data repository in Rapture has a unique name, a single class, and a single implementation. The implementation defines the underlying technology that manages the data, such as Oracle or Cassandra. Client applications do not need the implementation metadata; instead, they rely on URI references, as explained below.

Rapture uses a strict system for naming URIs, in which the end of the URI string always concludes with `authority/id`. The authority identifies the repository, and the id is a pointer to a specific entity within the repository. Depending on the class of data, the ID can be a simple string, or it can contain sub-parts. For example, the ID of an item in a series repository might be `prices/pork/04092014`, where `prices` is the name of the authority and `pork/04092014` is the ID for a specific item within one series.

Pipelining

When an activity takes place within Rapture, the platform uses a pipelining technique to determine which server is most appropriate for running the activity. Pipelining allows Rapture to dynamically scale the numbers and categories of servers as the load profile varies over time.

Pipeline task structure

All tasks submitted to the pipeline have a common set of fields, as shown in Table 1.

Table 1. Pipeline task fields

Field Name	Description
Priority	The priority of the task (lower is more important)
Category	A list of categories for this task (for routing)
Content Type	Determines how the message will be executed
Content	Depends on Content Type above
Context	The user context under which the task will be executed
Submission Time	When the task was added to the pipeline

The content type field contains information about the processing that will be needed for running this action. Table 2 lists the allowable values for this field.

Table 2. Content Type field values

Field Name	Description
text/plain	Any simple string; used for testing
application/vnd.rapture.reflex.script	The content is an embedded Reflex script, with parameters
application/vnd.rapture.reflex.ref	The content is a reference to a Rapture-hosted Reflex script, with parameters.
application/vnd.rapture.doc.save	The content is the display name for a document and the document's content. Both need to be saved.
application/vnd.rapture.operation	The content is a set of parameters for running an operation on Rapture.
application/vnd.rapture.audit	The content is an audit trail record that needs to be saved.

Pipeline configurations

All Rapture platforms have at least one kernel pipeline for handling transport activity. However, there can easily be many different pipelines within the same platform, each of which can be configured to a different, specific messaging transport. The configuration of a pipeline is stored in a system document repository for access by the Rapture

kernel. Physically, pipelines are hosted on a system designed to support message queues, such as RabbitMQ or MQSeries.

Events

The primary function of an event, once it's triggered, is to begin the execution of one or more scripts, which are generally Reflex scripts.

Event hooks

Events can be explicitly scheduled, or they can be connected to actions that happen at various "hook points" in the system. In most cases, the script references are connected to the hook asynchronously. These scripts are simply queued into the kernel pipeline to be run as early as possible. The context of the event is also passed as a parameter to the script.

In addition to – or instead of – asynchronous scripts, an event can be associated with a single *inline* script reference. Inline scripts are run immediately, as soon as the event occurs. They function as the equivalent of an interrupt processing routine, so the system user must take care to keep the execution as brief and efficient as possible. It is also important to avoid further calls that would trigger other events while the script is running.

As a shortcut, an event can also directly invoke a workflow, instead of requiring a developer to write a one-line script that starts the workflow.

Figure 3 shows the structure of the event hook.

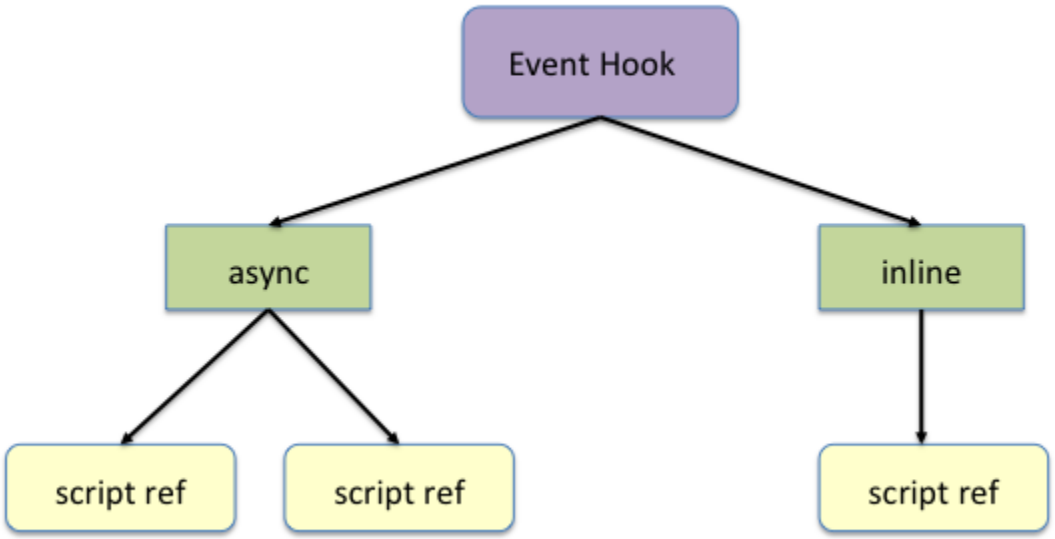

Figure 3. Event hook structure

Event flows

Figure 4 shows the steps that take place when an event is invoked.

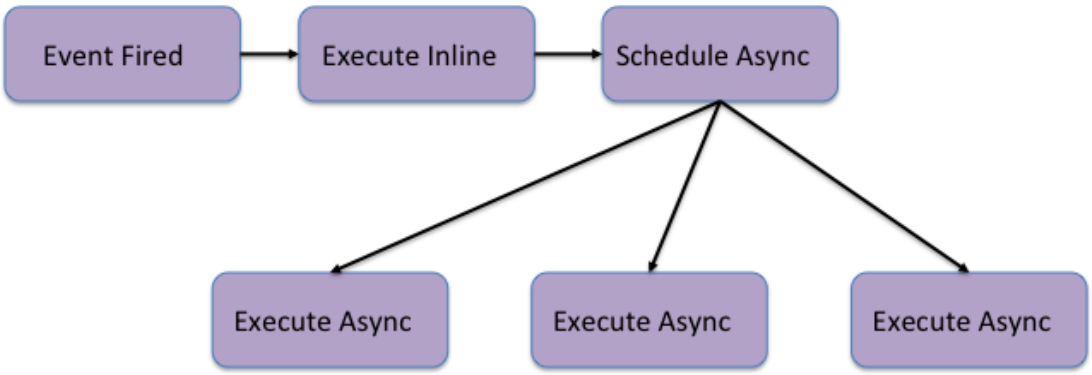

Figure 4. Event flow

Note that event information and event hooks are assigned to an arbitrary set of named events, located in the *event registry*. The purpose of the registry is to retain two distinct sets of information about an event: what happens when it's triggered, and what triggers it in the first place. This feature allows users to define custom events in addition to Rapture's built-in system events. The Rapture API contains calls to trigger user-defined events only.

Scripting

The Rapture platform includes a unique procedural scripting language called Reflex, which hooks into the Java Virtual Machine that Rapture uses.

NOTE: Reflex was designed to streamline many of the common tasks in Rapture, but it is not required for event-based scripts.

Reflex overview

For comprehensive information about the Reflex language, refer to Incapture's *Reflex Language Reference*.

Reflex is similar to Python in structure, and is a procedural language in that it supports author-defined functions. (In addition, Reflex is not object-oriented.) It is packaged with an emulator called ReflexRunner, so that scripts can be tested before being run on Rapture.

Data types in Reflex are mostly intuitive, including *string*, *number*, *list* (equivalent to arrays), *boolean*, and *map* (which is used for key-value pairs).

Operators unique to Reflex

Reflex supports all the standard boolean, arithmetic, index ([]) and ternary (?) operators. It also uses the standard keywords for flow control, including `if`, `while`, and `for`.

In addition, there are four special operators: push, pull, metapush, and metapull; whose symbols, respectively, are -->, <--, -->>, and <<--. The push and pull operators fetch data from Rapture and save it back. Metapush and metapull are similar, except that they fetch and save only the metadata tags. The following section demonstrates how push and pull might be used.

Simple Reflex examples

In this scenario, suppose you have a repository named test.config, and you are using it to store map data. You would define the structure and push it as follows:

```
config = {};
// Write the map data
config['Option1'] = true;
config['level'] = 42;


// Create a document
displayName = 'test.config/main';

// Write the map to the document

config --> displayName;
```

Next, in a different script, you would pull in the map data and use it to control the script's behavior:

```
appConfig <-- displayName;
if (appConfig['Option1']) do
  println("level is " + appConfig['level']);
else do
  println("Option1 is not set.");
end
```

The second example uses `println`, which is a built-in function in Reflex, to handle simple formatted printing. If all is well, the output from the second snippet of code would be: `level is 42`.

Decision Processes

A decision process works with a *Decision Packet*, which contains any data that needs to be associated with a given process. Such data includes status and progress information within the decision process and also can include references to documents, blobs, and audit logs.

The decision process itself is a directed graph of states, with one entry point and at least one terminal point. Each state has an associated script that determines whether the decision packet is ready to transition to the next state and, if there are multiple possibilities, which state it should transition to. There is also a script attached to the entry point, which executes before the process begins.

As a simple example of a decision process, a trade might require an authorization above a certain amount from the head of the trading desk. In this case, the decision packet would contain the trade information, and there would need to be three scripts associated with the decision process. One of these would check the amount, one would be run after the trade is approved, and one would be run after the trade is denied. If the trade amount exceeds the threshold, the decision process is halted, and the application using the decision process would need to run some additional code to notify both parties that an approval decision is needed.

For a more real-world example, consider a data capture scenario from an external source. The scripting steps in this process would be: (1) preparing the request, (2) connecting to the underlying system (for example, Bloomberg) and depositing the request, (3) waiting for request completion with a success or failure result, (4) downloading and processing the result if the request was successful, or (5) notifying the caller if the request failed.

The Rapture API contains calls for creating processes and packets, sending packets to processes, fetching and updating packet data, and registering approvals. This functionality makes scripting decision processes very straightforward.

Other Subsystems

This section covers the remaining Rapture subsystems from Figure 1.

Schedule

The Schedule subsystem, as its name implies, instructs Rapture to run one or more scripts in a *schedule entry* at a specific time, sending them to the kernel pipeline.

Schedule entries are stored in a tree format to manage the execution order of the scripts over time. Figure 5 shows an example with references to five scripts. Script 1 and Script 2 are executed serially, and once Script 2 has completed, Script 3 and Script 4 are sent to the pipeline to be run in parallel. Script 5 is not executed until ***both*** Script 3 and Script 4 have completed successfully.

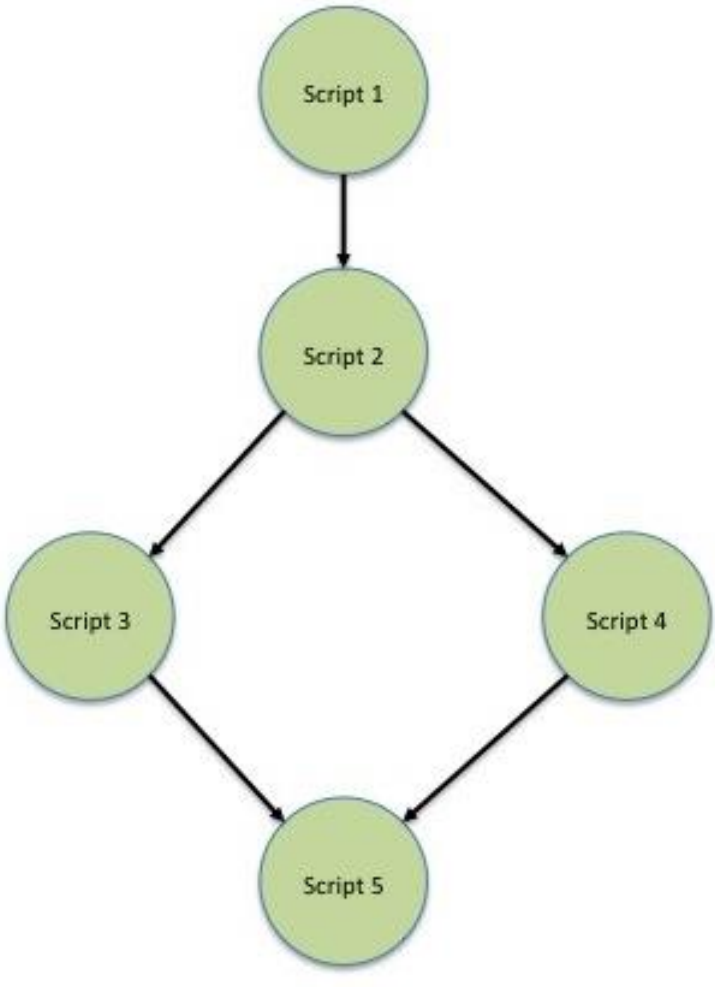

Figure 5. Schedule entry example

Entitlement

Every API call has at least one entitlement path associated with it. The call will not be run unless the user belongs to a group that has the API's entitlement path in its allow list. API calls can also have entitlement rules, which are wildcards that can generate multiple paths. Users are not assigned entitlement paths or rules directly; they can only be assigned to groups that have entitlements. When a user does not belong to a group that has access to an entitlement path, any attempt to make the corresponding API call will fail.

In practice, groups are often used for granting read-only access to some users and read-write access to others.

Audit

Audit functionality is available to Rapture system administrators. By default, there is a kernel audit trail for which the Rapture administrator can configure its storage and behavior. This audit trail is normally wired into the API invocation flow.

In addition, administrators can create their own audit trails, using the same API calls used by the kernel audit trail for generating its own records. In this case, either Reflex scripts or client applications would be used to generate the custom audit records.

Notification

In addition to the user-based notification channels discussed in the overview, the Rapture system maintains its own notification channels to report system-wide changes to the configuration or the structure of the Rapture platform. This feature allows for obsolete cached data structures to be purged before they can be erroneously used.

Extensions

Clients have access to several extensions to the Rapture platform.

Reflex plug-ins

Various plugins can allow Reflex scripts to be extended to local Java code or to remote services. These plugins need to be encapsulated in .jar files in the class path of the application that will run them.

The entry point to a plugin is a class that implements the `reflex.importer.Module` interface and is placed into the `reflex.Module` package. There are two possible ways to embed a function in Reflex: it can either be done with a keyhole call or by using reflection. The keyhole approach has better execution times, but reflection is more frequently used because the coding is less complex. Both overrides must be included, with one set to false and the other to true, as shown below:

```
@Override
public boolean handlesKeyhole() {
    return false;
}

@Override
public boolean canUseReflection() {
    return true;
}
```

In addition, there are three functions that pass configuration information to Rapture and need their own overrides. These functions are `configure`, `setReflexHandler`, and `setReflexDebugger`.

Finally, if keyhole calls are used, the class must override *keyholeCall*. Otherwise, reflection lets Reflex invoke any function that returns a *ReflexValue* type and that takes a parameter list of *ReflexValue*s.

As an example, the following plugin code implements a class called ReflexMath, which for brevity implements only a cosine function.

```
package reflex.module;

import java.util.List;

import reflex.IReflexHandler;
import reflex.ReflexException;
import reflex.debug.IReflexDebugger;
import reflex.importer.Module;
import reflex.value.ReflexValue;
import reflex.value.internal.ReflexVoidValue;

public class ReflexMath implements Module {
```

```
    private double getDP(List<ReflexValue> params) {
        if (params.size() == 1) {
            if (params.get(0).isNumber()) {
                return params.get(0).asDouble();
            }
        }
        throw new ReflexException(-1, "Cannot retrieve numeric
first argument in math call");
    }

    private double getDP(List<ReflexValue> params, int position)
{

    @Override
    public ReflexValue keyholeCall(String name, List<ReflexValue>
parameters) {
        return new ReflexVoidValue();
    }

    @Override
    public boolean handlesKeyhole() {
        return false;
    }

    @Override
    public boolean canUseReflection() {
        return true;
    }

    @Override
    public void configure(List<ReflexValue> parameters) {
    }

    @Override
    public void setReflexHandler(IReflexHandler handler) {
    }

    @Override
    public void setReflexDebugger(IReflexDebugger debugger) {

    }

    public ReflexValue cos(List<ReflexValue> params) {
        return new ReflexValue(Math.cos(getDP(params)));
    }
}
```

To use `ReflexMath` in another Reflex script, you could do something like this:

```
import ReflexMath as math;

println("cos(0.1) = " + $math.cos(0.1));
```

Storage extensions

Copies of the pre-built storage implementation can be customized. Contact Incapture for additional details.

Custom APIs

Clients have access to the same tools that were used for creating the Rapture API. As a result, it is possible for clients to create their own high-level APIs with automatic client-side code generation for invoking calls.

For client applications in financial services, some common applications that could use the API include a Value at Risk (VaR) system for a broker-dealer; a system for order construction, programmed trading, and position management used in a hedge fund; or an equity research management system.

Beyond financial service, the Rapture platform offers many other possibilities for useful applications and big data management. For instance, a system for collecting multiple sensor readings from city locations, vehicles, or even microsurgical devices can be used with the Rapture platform to handle complex decisions in real time.