Eventing 6.1 Specification

Operations

The following concepts shall be exposed to end users through UI with below semantics. Admin operations are accessible through UI, and via couchbase-cli.

Deploy

This operation activates a function. Source validations are performed, and only valid functions can be deployed. Deployment transpiles the code and creates the executable artifacts. The source code of an activated function cannot be edited. Unless a function is in deployed state, it will not receive or process any events. Deployment creates necessary metadata, spawns worker processes, calculates initial partitions, and initiates checkpointing of processed stream data.

Deployment has two variations:

Deploy from Start

This choice currently affects only DCP observer. In this choice, the Function will see a deduplicated history of all documents, ending with the current value of each document. Hence, the Function will see every document in the bucket at least once in this choice.

Deploy from Now

This choice currently affects only DCP observer. This choice causes functions to start observing mutations from current time. In other words, the Function will see only document mutations that occur after it is deployed.

Undeploy

This operation causes the function to stop processing events of all types and shuts down the worker processes associated with the function. It deletes all timers created by the function being undeployed and their context documents. It releases any runtime resources acquired by the function. Functions in undeployed state allow code to be edited. Newly created functions start in Undeployed state.

Pause

This stops all processing associated with a function including timer callbacks. A function in paused state can be edited. Functions in Paused state can be either Resumed or Undeployed.

Resume

This continues processing of a function that was previously Paused. The backlog of mutations that occurred when the function was paused will now be processed. The backlog of timers that came due when the function was paused will now fire. Depending on the system capacity and how long the function was paused, clearing the backlog may take some time before Function moves on to current mutations and timers.

It is the responsibility of the user that any code edits made to a Function when it was in Paused state is compatible with the artifacts and timers registered by the prior version of the function.

Delete

When a function is deleted, the source code implementing the function, all timers, all processing checkpoints and other artifacts in metadata provider is purged. A future function by the same name has no relation to a prior deleted function of the same name. Only undeployed functions can be deleted.

Debug

Debug is a special flag on a function that causes the next event instance received by the function be trapped and sent to a separate v8 worker with debugging enabled. The debug worker pauses the trapped event processing and opens an TCP port and generates a Chrome devtools URL with a session cookie that can be used to control the debug worker. All other events, except the trapped event instance, continue unencumbered. If the debugged event instance completes execution, another event instance is trapped for debugging, and this continues till debugging is stopped, at which point any trapped instance runs to completion and debug worker passivates.

Debugging is convenience feature intended to help during function development and should not be used in production environments. Debugger does not provide correctness or functionality guarantees.

Objects

Binding

A binding is a construct that allows separating environment specific variables (example: bucket names, external endpoint URLs, credentials) from the function source code. It provides a level of indirection between environment specific artifacts to symbolic names, to help moving a function definition from development to production environments without changing code. Binding names must be valid JavaScript identifiers and must not conflict any built-in types.

Bucket Bindings

Bucket bindings allow JavaScript functions to access Couchbase KV buckets. The buckets are then accessible by the bound name as a JavaScript map in the global space of the handler.

Read Only Bindings

A binding with access level of "Read Only" allows reading documents from the bucket, but cannot be used to write (create, update or delete) documents in such a bucket. Attempting to do so will throw a runtime exception.

Read-Write Bindings

A binding with access level of "Read Write" allows both reading and writing (create, update, delete) of documents in the bucket.

Recursion

When a Function manipulates documents in a bucket that serves as the source of mutations to this or any other Function, a write originated by a Function will cause a mutation to be seen by itself or another function. We call these potentially recursive mutations, because depending on the code and configuration, it can cause recursion of mutation between the bucket and the function.

Mutual Recursion

When functions manipulate buckets that are the source of mutations to other Functions, mutual recursions can result. These are difficult to detect and suppress, and so as a general rule, developers are discouraged (though not prohibited) from chaining functions. If functions are manipulating buckets that are source of other functions, extreme caution must be exercised to ensure mutual recursions does not result before deploying the function.

Direct Self-Recursion

A special case of this is when a function handler chooses to create a Read-Write binding to its own source bucket. In such a setup, every write by the Function to the source bucket will cause a mutation back to the Function for the very same write it just executed. As such self-recursion is of little value, but the ability to mutate documents in the source bucket is useful for document enrichment use cases, the eventing framework detects and suppresses such direct self-recursive mutations. Due to this built-in support, this configuration does not require as much caution before using as general recursive functions.

URL Bindings

These bindings are utilized by the curl language construct to access external resources. The binding specifies the endpoint, the protocol (http/https), and credentials if necessary. Cookie support can be enabled via the binding if desired when accessing trusted remote nodes. When a URL binding limits access through to be the URL specified or descendants of it. The target of a URL binding should be not be a node that belongs to the Couchbase cluster.

Language Constructs

In general, functions inherit support for most ECMAScript constructs by virtue of using Google v8 as the execution container. However, to support ability to automatically shard and scale the function execution, we need to remove a number of capabilities, and to make the language utilize the server environment effectively, we introduce a few new constructs.

Language Constructs - Removed

The following notable JavaScript constructs cannot be used in Functions.

Global State

Functions do not allow global variables. All state must be saved and retrieved from persistence providers. At present, the only available persistence provider is the KV provider, and so all global state is contained to the KV bucket(s) made available to the function via bindings. This restriction is necessary to enable function logic to remain agnostic of rebalance.

```
1.  var count = 0;                          // Not allowed - global variable.
2.  function OnUpdate(doc, meta) {
3.      count++;
4.  }
```

Asynchrony

Asynchrony, and in particular, asynchronous callback can and often must retain access to parent scope to be useful. This forms a node specific long running state which prevents from capturing entire long running state in persistence providers. So, function handlers are restricted to run as short running straight line code without sleeps and wakeups. We do however add back limited asynchrony via time observers (but these are designed to not make the state node specific).

```
1.  function OnUpdate(doc, meta) {
2.    setTimeout(function(){}, 300);     // Not allowed - asynchronous flow.
3.  }
```

Browser and Other Extensions

As functions do not execute in context of a browser, the extensions browsers add to the core language, such as window methods, DOM events etc. are not available. A limited subset is added back (such as function timers in lieu of setTimeout, and curl calls in lieu of XHR).

```
4.  function OnUpdate(doc, meta) {
5.    var rpc = window.XMLHttpRequest();  // Not allowed - browser extension.
6.  }
```

In addition, other v8 embedders have introduced extensions such as require() in Node.js which are currently not adopted by functions, but may be done so in future where such extensions play well in the sandbox required of functions.

Language Constructs - Added

The following constructs are added into the functions JavaScript.

Bucket Accessors

Couchbase buckets, when bound to a function, appears as a global JavaScript map. Map get, set and delete are mapped to KV get, set and delete respectively. Other advanced KV operations will be available as member functions on the map object.

```
function OnUpdate(doc, meta) {
  // Assuming 'dest' is a bucket alias binding
  var val = dest[meta.id];         // this is a bucket GET operation.
  dest[val.parent] = {"status":3}; // this is a bucket SET operation.
  delete dest[meta.id];            // this is a bucket DEL operation.
}
```

Logging

An additional function, log() has been introduced to the language, which allows handlers to log messages. These messages go into the eventing data directory and do not contain any system log messages. The function takes a string to write to the file. If non-string types are passed, a best effort string representation will be logged, but the format of these may change over time.

```
function OnUpdate(doc, meta) {
    log("Now processing: " + meta.id);
}
```

N1QL

Top level N1QL keywords, such as SELECT, UPDATE, INSERT, are available as keywords in functions. Operations that return values such as SELECT are accessible through a special iterator on which the *for (var <row> of <iterator>)* looping construct has been defined. This restricted looping construct allows us to support query result streaming, and automatic query cancellation when the iterator goes out of scope.

JavaScript variables can be referred by N1QL statements using *$<variable>* syntax. Such parameters will be substituted with the corresponding JavaScript variable's runtime value using N1QL named parameters substitution facility.

```
function OnUpdate(doc, meta) {
    var strong = 70;
    var results =
       SELECT *                  // N1QL queries are embedded directly.
       FROM `beer-samples`       // Token escaping is standard N1QL style.
       WHERE abv > $strong;      // Local variable reference using $ syntax.
    for (var beer of results) {  // Stream results using 'for' iterator.
        log(beer);
        break;                   // Cancel streaming query by breaking out.
    }
}
```

Timers

Functions can register to observe wall clock time events. Timers are sharded across eventing nodes, and so are scalable. For this reason, there is no guarantee that a timer will fire on the same node on which it was registered or that relative ordering between any two timers will be maintained. Timers only guarantee that they will fire at or after the specified time.

When using timers, it is required that all nodes of the cluster are synchronized at computer startup, and periodically afterwards using a clock synchronization tool like NTP.

Creating a Timer

Timers are created as follows:

```
createTimer(callback, date, reference, context)
```

callback

This function is called when the timer fires. The callback function must be a top-level function that takes a single argument, the context (see below).

date

This is a JavaScript Date object representing the time for the timer to fire. The date of a timer must always be in future when the timer is created, otherwise the behavior is unspecified.

reference

This is a unique string that must be passed in to help identify the timer that is being created. References are always scoped to the function and callback they are used with and need to be unique only within this scope. If multiple timers are created with the same unique reference, old timers with the same unique reference are implicitly cancelled.

context

This is any JavaScript object that can be serialized. The context specified when a timer is created is passed to the callback function when the timer fires. The default maximum size for Context objects is 1kB. Larger objects would typically be stored as bucket objects, and document key can be passed as context.

Cancelling a Timer [WIP]

Timers can be cancelled as follows:

```
cancelTimer(callback, reference)
```

Timers can be cancelled by the function that created them by passing the same callback and reference that was used when it was created to the cancel call. If no such timer exists, or the timer has already fired, the cancel call has no effect.

cURL

The cURL feature provides a way of interacting with external entities using HTTP from event handlers.

API

response_object = **curl(**method, binding, [request_object]**)**

Arguments

The curl call accepts the following parameters:

method

The HTTP method of the cURL request. Must be a string having one of the following values: GET | POST | PUT | HEAD | DELETE.

binding

The cURL binding that represents the http endpoint URL that will be accessed by this call.

request_object

This parameter captures the request and related information. The request_object is a JavaScript object having the following keys:

headers

Optional. A JavaScript Object of key-value pairs with key representing the header name and value representing the header content. Both key and value must be strings.

body

A JavaScript variable representing the content of the request body. See below for details on how various JavaScript variable types are marshalled to form the HTTP request.

encoding

Optional. A directive on how to encode the body. A string having one of below values: FORM | JSON | TEXT | BINARY.

path

The sub-path the request is made. This must be a string and will be appended to the URL specified on the binding object.

params

This must be a JavaScript Object of key-value pairs. Keys must be strings, and values must be string, number or boolean. These will be URL encoded as HTTP request parameters and appended to the request URL.

Return value (response_object)

The returned value from the cURL call which captures the response of the remote HTTP server to the request made. This is a JavaScript Object containing the following fields:

body

A JavaScript variable representing the content of the response body. See below for details on how the response is unmarshalled into various JavaScript variable types.

status

The numeric HTTP status code.

headers

A JavaScript Object of key-value pairs with key representing the header name and value representing the header content. Both key and value will be strings.

Exceptions Thrown

When an unexpected error occurs, a JavaScript exception of type CurlError inheriting from the JavaScript Error class will be thrown.

Bindings

To access a HTTP server using cURL, the handler needs to declare a URL binding and pass the alias of the binding to curl() calls. The binding specifies the remote URL to be accessed and all calls made using such a binding are limited to descendants of the URL specified in the binding.

HTTPS is used when the URL specifies the *https://* prefix. Such a link uses https for encryption of contents, and if enabled, verifies the server certificate using the underlying OS support for server certificate verification. Client certificates are not currently supported. Any include/exclude cipher rules setup at Couchbase Server level will apply here as well.

The binding may also specify the authentication mechanism and credentials to use. Basic, Digest and Bearer authentication methods are supported. It is strongly recommended that when authentication is used, the binding uses only https protocol to ensure credentials are encrypted when transmitted.

Cookie support may be enabled at binding level if desired when accessing controlled and trusted endpoints.

Example

In the below example, a cURL request is created to the specified binding *profile_svc_binding* with the sub-URL */person* with URL parameters *action* and *id* and the body being a JSON object. The response is a JSON object and is seen containing a field *profile_id*. In this example, the request is automatically encoded as application/json and response is automatically parsed from JSON response, as no explicit encoding is specified.

```
1.  var request = {
2.    path: '/person',
3.    params: {
4.      'action': 'create',
5.      'id': 23012
6.    },
7.    body: {
8.      'name': 'John Smith',
9.      'age': 25,
10.     'state': 'CA',
11.     'country': 'US',
12.   }
13. };
14.
15. var response = curl('POST', profile_svc_binding, request);
16. if (response.status == 200) {
17.   var profile_id = response.body.profile_id;
18.   log("Successfully created profile " + profile_id);
19. }
20.
```

Request marshalling

The framework attempts to automatically encode JS objects to the most appropriate encoding and generate the appropriate Content-Type header. Such automatic request marshalling is controlled by the type of JavaScript object passed into the request *body* parameter and optionally, the value set for request *encoding* parameter.

Below table shows the encoding and Content-Type chosen based on JS object passed:

JS object passed to the *body* param	**Value passed for *encoding* param**	**Encoding used for request body**	**Content-Type header sent (unless overridden by *headers* param)**
JS String	(not specified)	UTF-8	text/plain
JS Object	(not specified)	JSON	application/json
JS ArrayBuffer	(not specified)	Raw Bytes	application/octet-stream
JS String	TEXT	UTF-8	text/plain
JS Object	TEXT	(disallowed)	(disallowed)
JS ArrayBuffer	TEXT	(disallowed)	(disallowed)
JS String	FORM	URL Encoding	application/x-www-form-urlencoded
JS Object	FORM	URL Encoding	application/x-www-form-urlencoded
JS ArrayBuffer	FORM	(disallowed)	(disallowed)
JS String	JSON	JSON	application/json
JS Object	JSON	JSON	application/json
JS ArrayBuffer	JSON	(disallowed)	(disallowed)
JS String	BINARY	UTF-8	application/octet-stream
JS Object	BINARY	(disallowed)	(disallowed)
JS ArrayBuffer	BINARY	Raw Bytes	application/octet-stream

Users who wish to utilize custom encoding can do so by specifying an appropriate Content-Type using the *headers* parameter of the request object and passing the custom encoded object as an ArrayBuffer as the *body* parameter of the request.

Response unmarshalling

Response object from the remote is automatically unmarshalled if the response contains a recognized Content-Type header. The following table identifies the action used to unmarshall responses:

Content-Type specified by response	**Unmarshalling action**	**Response *body* param**
text/plain	Convert to string as UTF-8	JS string
application/json	JSON.parse()	JS Object
application/x-www-form-urlencoded	decodeURI()	JS Object or JS String
application/octet-stream	Store raw bytes	JS ArrayBuffer
(Content-Type not listed above)	Store raw bytes	JS ArrayBuffer

(Content-Type header missing)	Store raw bytes	JS ArrayBuffer

Session handling

Cookie support is turned off by default on a cURL binding. So, no cookies will be accepted from the remote server. Cookies can be enabled if accessing a controlled and trusted endpoint. If enabled, cookies are accepted and stored in-memory of the worker object, scoped to the binding object.

Note that eventing utilizes multiple workers and multiple HTTP cURL sessions and so a handler cannot rely on all requests executing on the same HTTP session. It can rely on issued cookies being presented on subsequent requests only within the duration of a single eventing handler invocation.

Built-in Functions

crc64

This function calculates the CRC64 hash of an object using the ISO polynomial. The function takes one parameter, the object to checksum, and this can be any JavaScript object that can be encoded to JSON. The hash is returned as a string (because JavaScript numeric types offers only 53-bit precision). Note that the hash is sensitive to ordering of parameters in case of map objects.

```
1.  function OnUpdate(doc, meta) {
2.    var crc_str = crc64(doc);
3.    ...
4.  }
```

Handler Signatures

The following event handlers are available:

Insert/Update Handler

The insert/update handler gets called when a document is created or modified. Two major limitations exist. First, if a document is modified several times in a short duration, the calls may be coalesced into a single event due to deduplication. Second, it is not possible to discern between Create and Update operations. Both limitations arise due to KV engine design choices and may be revisited in the future.

```
21. function OnUpdate(doc, meta) {
22.   if (doc.type == 'order' && doc.value > 5000) {
23.     phoneverify[meta.id] = doc.customer;
24.   }
25. }
```

Delete Handler

The delete handler gets called when a document is created or modified. Two major limitations exist. First, it is not possible to discern between Expiration and Delete operation. Second, it is not possible to get the value of the document that was just deleted or expired. Both limitations arise due to KV engine design choices and may be revisited in the future.

```
function OnDelete(meta) {
  var addr = meta.id;
  var res = SELECT id from orders WHERE shipaddr = $addr;
  for (var id of res) {
    log("Address invalidated for pending order: " + id);
  }
}
```

Terminology

Function

A function is a collection of handlers implementing a composite business functionality. Resources are managed at function level (or above) and the state of all handlers is scoped by the containing function.

Handler

A handler is a piece of code reacting a specified event. One or more handlers together constitute a function. A handler is stateless short running piece of code that must execute from start to end prior to a specified timeout duration.

Statelessness

The characteristic that any persistent state of a function is captured in entirety by the below, and all states that appears on the execution stack are ephemeral.

1. The metadata bucket (which will eventually be a system collection)
2. The documents being observed
3. The storage providers bound to the function

Deduplication

Couchbase does not store every version of a document permanently. Hence, when a Function asks for mutation history of a document, it sees a truncated history of the document. However, the final state of a document is always present in all such histories (as the current state is always available in the database).

Similarly, the KV data engine deduplicates multiple mutations made to any individual document rapidly in succession, to ensure highest possible performance. So, when a document mutates rapidly, Functions may not see all intermediate states, but in all cases, will see the final state of the document.

Recursive Mutation

An abbreviation of convenience of the term *Potentially Recursive Mutation*. When a Function manipulates documents in a bucket that serves as the source of mutations to this or any other Function, a write originated by a Function will cause a mutation to be seen by itself or another function. These are called potentially recursive mutations.