COUCHBASE EVENTING SPECIFICATION

Introduction

Couchbase Eventing offers the ability to write programmatic logic that can respond to various events within Couchbase Server. The intent of the framework is to capture the business logic, and automatically parallelize it across the cluster to ensure high throughput and scalability without requiring the handler logic to code these aspects explicitly. Eventing is an assignable MDS role and so, eventing can run on any specified set of nodes in a cluster. Couchbase Eventing was first introduced in 5.5.0 and continues to be in active development. Couchbase Eventing is a capability of Couchbase Server Enterprise Edition (EE).

Handler Signatures

Eventing framework calls the following JavaScript functions as entry points to the handler.

Insert/Update Handler

The *OnUpdate* 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.

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

Delete Handler

The *OnDelete* handler gets called when a document is deleted (either directly, such as a N1QL or KV DELETE operation, or indirectly via expiry pager deleting expired documents). The function provides two[1] arguments, first being metadata of the document that was deleted (*meta* in the function below), and the second (*options* in the example below) being additional information. Both are JavaScript maps.

The 'expired' option

The second argument to the OnDelete function (which is a JavaScript map object) contains a boolean entry having key name '*expired*' that indicates if the deletion was due to a document expiration.

```
function OnDelete(meta, options) {
  if (options.expired) log("Document expired", meta.id);
  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);
```

[1] Also see note in "Language Change History" section regarding this function signature change

```
  }
}
```

OnDeploy Handler [Currently Unimplemented]

The *OnDeploy* handler is invoked once when the handler has deployed. This callback will fire only once per deployment and will run on only one eventing node in the cluster. The *OnDeploy* handler is the first function to run for the given handler, and all other function calls on all nodes wait for the OnDeploy handler to complete. OnDeploy handler is subject to normal handler timeout.

```
function OnDeploy() {
  log("Function was deployed:", new Date());
}
```

Operations

The following operations are exposed through the UI, couchbase-cli and REST APIs.

Deploy

This operation activates a handler. Source validations are performed, and only valid handlers can be deployed. Deployment transpiles the code and creates the executable artifacts. The source code of an activated handler cannot be edited. Unless a handler 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 for DCP observer has two variations:

Deploy from Start

The Handler will see a deduplicated history of all documents, ending with the current value of each document. Hence, the Handler will see every document in the bucket at least once.

Deploy from Now

The handlers will see mutations from current time. In other words, the Handler will see only documents that mutate after it is deployed.

Undeploy

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

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

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

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

Delete
When a handler is deleted, the source code implementing the handler is purged. Only handlers in undeployed state can be deleted. A future handler by the same name has no relation to a prior deleted handler of the same name.

Debug
Debug is a special flag on a handler that causes the next event instance received by the handler 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 handler development and should not be used in production environments. Debugger does not provide correctness or functionality guarantees. Debugger is disabled by default and must be enabled in the global Eventing Settings page before any handler can be debugged.

Objects

Binding
A binding is a construct that allows separating environment specific variables (example: bucket names, external endpoint URLs, credentials) from the handler source code. It provides a level of indirection between environment specific artifacts to symbolic names, to help moving a handler 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 handlers 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 Handler manipulates documents in a bucket that serves as the source of mutations to this or any other Handler, a write originated by a Handler will cause a mutation to be seen by itself or another handler. We call these potentially recursive mutations, because depending on the code and configuration, it can cause recursion of mutation between the bucket and the handler.

Mutual Recursion

When handlers manipulate buckets that are the source of mutations to other Handlers, 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 handlers. If handlers are manipulating buckets that are source of other handlers, extreme caution must be exercised to ensure mutual recursions does not result before deploying the handler.

Direct Self-Recursion

A special case of this is when a handler handler chooses to create a Read-Write binding to its own source bucket. In such a setup, every write by the Handler to the source bucket will cause a mutation back to the Handler 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 handlers.

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, handlers 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 handler 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 Handlers.

Global State

Handlers 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 handler via bindings. This restriction is necessary to enable handler logic to remain agnostic of rebalance.

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

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).

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

Browser and Other Extensions

As handlers 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).

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

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

Language Constructs - Added

The following constructs are added into the handlers JavaScript.

Bucket Accessors

Couchbase buckets, when bound to a handler, 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 handlers 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.
}
```

Get operation (operator [] applied on a bucket binding and used as a value expression)

This fetches the corresponding object from the KV bucket the variable is bound to, and returns the parsed JSON value as a JavaScript object. Fetching a non-existent[2] object from a bucket will return JavaScript *undefined* value. This operation throws an exception if the underlying bucket GET operation fails with an unexpected error.

Set operation (operator [] appearing on left of = assignment statement)

This sets the provided JavaScript value into the KV bucket the variable is bound to, replacing any existing value with the specified key. This operation throws an exception if the underlying bucket SET operation fails with an unexpected error.

Delete operation (operator [] appearing after JavaScript delete keyword*)*

This deletes the provided key from the KV bucket the variable is bound to. If the object does not exist, this call is treated as a no-op. This operation throws an exception if the underlying bucket DELETE operation fails with an unexpected error.

Advanced Bucket Accessors

It is possible to access other advanced KV functionality using the following built-in operators. These utilize the same bucket bindings defined in the handler, but exposer richer set of options and operators.

[2] Also see note in "Language Change History" section regarding behavior change

Note that in this section, CAS values where they appear are of JavaScript *string* types (and not *number* type). This is because v8 JavaScript *number* type is accurate only to 53 bits, while CAS utilizes all 64-bits. Hence, it is both received and returned as a *string* type to ensure fidelity is maintained when passing CAS values from *get()* operations to *set()* operations.

Advanced GET:

result = couchbase.get(binding, meta)

This operation allows reading a document from the bucket with ability to specify the CAS value to be matched during the read.

binding
The name of the binding that references the target bucket. The binding can have access level of *“read”* or *“read/write”*.

meta (type: Object)
The positional parameter (denoted by “*meta”* in the prototype above) represents the metadata of the operation. At minimum, the document key must be specified in this object.

meta.id (type: string)
The key of the document to be used for the operation. This is a mandatory parameter and must be of JavaScript *string* type.

result – the return value (type: Object)
The return object indicates success/failure of the operation, and the data fetched if successful, or the error details if failure.

result.success (type: boolean)
This field indicates if the operation was successful or not. It is always present in the return object.

result.meta (type: Object)
This field is present only if the operation succeeded. It contains metadata about the object that was fetched.

result.meta.id (type: string)
The key of the document that was fetched by this operation.

result.meta.cas (type: string)
The CAS value of the document that was fetched by this operation.

result.meta.expiry_date (type: Date)
The expiration date of the document. If no expiration is set on the document, this field will be absent.

result.doc (type: string, number, boolean, null, Object or Array)
If the operation is successful, this field contains the content of the requested document.

result.error (type: Object)
This field is populated if the operation failed. The contents of the error varies based on the type of error encountered, and commonly occurring fields are documented below.

result.error.key_not_found (type: boolean)
If present and set to true, this indicates that the operation failed because the specified key did not exist in the bucket.

result.error.code (type: number)
If present, the code of the SDK error that triggered this operation to fail. This is typically an internal numeric code.

result.error.name (type: string)
If present, the key is a stable token indicating the error that SDK encountered that caused this operation to fail. Error keys are stable over multiple releases and may be safely compared.

result.error.desc (type: string)
If present, a human readable description of the error that occurred. The description is for diagnostics and logging purposes only and may change over time. No programmatic logic should be tied to specific contents from this field.

exceptions
This API indicates errors via the error object in the return value. Exceptions are thrown only during system failure conditions.

Advanced INSERT:
result = couchbase.insert(binding, meta, doc)

This operation allows creating a fresh document in the bucket. This will fail if the document with the specified key already exists. An expiration may be optionally specified.

binding
The name of the binding that references the target bucket. The binding must have access level of *“read/write”*.

meta (type: Object)
The positional parameter (denoted by “*meta”* in the prototype above) represents the metadata of the operation. The document key must be specified in this *meta* object.

meta.id (type: string)
The key of the document to be used for the operation. This is a mandatory parameter and must be of JavaScript *string* type.

meta.expiry_date (type: Date)
This is an optional parameter, and if specified must be of JavaScript *Date* object type. The document will be marked to expire at the specified time. If no *expiry_date* is passed, no expiration will be set on the document.

doc (type: any JSON serializable)
This is the document content for the operation. This can be any JavaScript object that can be serialized to JSON (i.e., number, string, boolean, null, object and array).

result – the return value (type: Object)
The return object indicates success/failure of the operation, and the metadata of the operation.

result.success (type: boolean)
This field indicates if the operation was successful or not. It is always present in the return object.

result.meta (type: Object)
This field is present only if the operation succeeded. It contains metadata about the object that was inserted.

result.meta.id (type: string)
The key of the document that was inserted by this operation.

result.meta.cas (type: string)
The CAS value of the document that was created by this operation.

result.meta.expiry_date (type: Date)
The expiration field of the document, if one was set. If no expiration is set on the document, this field will be absent.

result.error (type: Object)
This field is populated if the operation failed. The contents of the error varies based on the type of error encountered, and commonly occurring fields are documented below.

result.error.key_already_exists (type: boolean)
If present and set to true, this indicates that the operation failed because the specified key already existed, and so the insertion operation failed.

result.error.code (type: number)
If present, the code of the SDK error that triggered this operation to fail. This is typically an internal numeric code.

result.error.name (type: string)
If present, the key is a stable token indicating the error that SDK encountered that caused this operation to fail. Error keys are stable over multiple releases and may be safely compared.

result.error.desc (type: string)
If present, a human readable description of the error that occurred. The description is for diagnostics and logging purposes only and may change over time. No programmatic logic should be tied to specific contents from this field.

exceptions
This API indicates errors via the error object in the return value. Exceptions are thrown only during system failure conditions.

Advanced UPSERT:

result = couchbase.upsert(binding, meta, doc)

This operation allows updating an existing document in the bucket, or if absent, creating a fresh document with the specified key. The operation allows specifying CAS value that must be matched as a pre-condition before proceeding with the operation. It also allows specifying an expiration time to be set on the document.

binding

The name of the binding that references the target bucket. The binding must have access level of *"read/write"*.

meta (type: Object)

The positional parameter (denoted by "*meta*" in the prototype above) represents the metadata of the operation. At minimum, the document key must be specified in this object.

meta.id (type: string)

The key of the document to be used for the operation. This is a mandatory parameter and must be of JavaScript *string* type.

meta.cas (type: string)

This is an optional parameter that specifies the CAS value to be used as a pre-condition for the operation. If the document's CAS value does not match the CAS value specified here, the operation will fail, setting the parameter *cas_mismatch* to *true* in the error object of the response object.

meta.expiry_date (type: Date)

This is an optional parameter. If specified, it must be of JavaScript *Date* object type. The document created or updated by this operation will be marked to expire at the specified time. If no expiration is provided, and if the document had a prior expiration set, the prior expiration will be cleared.

doc (type: any JSON serializable)

This is the document content for the operation. This can be any JavaScript object that can be serialized to JSON (i.e., number, string, boolean, null, object and array).

result – the return value (type: Object)

The return object indicates success/failure of the operation, and the metadata of the operation.

result.success (type: boolean)
This field indicates if the operation was successful or not. It is always present in the return object.

result.meta (type: Object)
This field is present only if the operation succeeded. It contains metadata about the object that was inserted or updated.

result.meta.id (type: string)
The key of the document that was inserted or updated by this operation.

result.meta.cas (type: string)
The CAS value of the document that was created or updated by this operation.

result.meta.expiry_date (type: Date)
The expiration field of the document, if one was set. If no expiration is set on the document, this field will be absent.

result.error (type: Object)
This field is populated if the operation failed. The contents of the error varies based on the type of error encountered, and commonly occurring fields are documented below.

result.error.cas_mismatch (type: boolean)
If present and set to true, this indicates that the operation failed because a CAS value was specified, and the CAS value on the object did not match the requested value.

result.error.code (type: number)
If present, the code of the SDK error that triggered this operation to fail. This is typically an internal numeric code.

result.error.name (type: string)
If present, the key is a stable token indicating the error that SDK encountered that caused this operation to fail. Error keys are stable over multiple releases and may be safely compared.

result.error.desc (type: string)
If present, a human readable description of the error that occurred. The description is for diagnostics and logging purposes

only and may change over time. No programmatic logic should be tied to specific contents from this field.

exceptions

This API indicates errors via the error object in the return value. Exceptions are thrown only during system failure conditions.

Advanced DELETE:

results = couchbase.delete(binding, meta)

This operation allows deleting a document in the bucket specified by key. Optionally, a CAS value may be specified which will be matched as a pre-condition to proceed with the operation.

binding

The name of the binding that references the target bucket. The binding must have access level of *"read/write"*.

meta (type: Object)

The positional parameter (denoted by "*meta"* in the prototype above) represents the metadata of the operation. At minimum, the document key must be specified in this object.

meta.id (type: string)

The key of the document to be used for the operation. This is a mandatory parameter and must be of JavaScript *string* type.

meta.cas (type: string)

This is an optional parameter that specifies the CAS value to be used as a pre-condition for the operation. If the document's CAS value does not match the CAS value specified here, the operation will fail, setting the parameter *cas_mismatch* to *true* in the error object of the response object.

result – the return value (type: Object)

The return object indicates success/failure of the operation, and the metadata of the operation.

result.success (type: boolean)

This field indicates if the operation was successful or not. It is always present in the return object.

result.meta (type: Object)
This field is present only if the operation succeeded. It contains metadata about the object that was deleted.

result.meta.id (type: string)
The key of the document that was deleted by this operation.

result.error (type: Object)
This field is populated if the operation failed. The contents of the error varies based on the type of error encountered, and commonly occurring fields are documented below.

result.error.key_not_found (type: boolean)
If present and set to true, this indicates that the operation failed because the specified key did not exist in the bucket.

result.error.cas_mismatch (type: boolean)
If present and set to true, this indicates that the operation failed because a CAS value was specified, and the CAS value on the object did not match the requested value.

result.error.code (type: number)
If present, the code of the SDK error that triggered this operation to fail. This is typically an internal numeric code.

result.error.name (type: string)
If present, the key is a stable token indicating the error that SDK encountered that caused this operation to fail. Error keys are stable over multiple releases and may be safely compared.

result.error.desc (type: string)
If present, a human readable description of the error that occurred. The description is for diagnostics and logging purposes only and may change over time. No programmatic logic should be tied to specific contents from this field.

exceptions
This API indicates errors via the error object in the return value. Exceptions are thrown only during system failure conditions.

Advanced INCREMENT:

results = couchbase.increment(binding, meta)

This operation atomically increments the field "*count*" in the specified document. The document must have the below structure:

```
{"count": 23} // 23 is the current counter value
```

The *increment* operation returns the post-increment value.

If the specified counter document does not exist, one is created with `count` value as 0 and the structure noted above. And so, the first returned value will be 1.

JavaScript numbers are accurate only till 53-bits. Hence, using *get()* followed by an *upsert()* with CAS rather than *increment()* is recommended when using counts that may go beyond $\pm 2^{53}$.

Due to limitations in KV engine API, this operation cannot currently manipulate full document counters.

binding

The name of the binding that references the target bucket. The binding must have access level of *"read/write"*.

meta (type: Object)

The positional parameter (denoted by "*meta"* in the prototype above) represents the metadata of the operation. At minimum, the document key must be specified in this object.

meta.id (type:string)

The key of the document to be used for the operation. This is a mandatory parameter and must be of JavaScript *string* type.

result – the return value (type: Object)

The return object indicates success/failure of the operation, and the metadata of the operation.

result.success (type: boolean)

This field indicates if the operation was successful or not. It is always present in the return object.

result.meta (type: Object)
This field is present only if the operation succeeded. It contains metadata about the counter that was incremented (or created and incremented).

result.meta.id (type: string)
The key of the document that was inserted by this operation.

result.doc.count (type: number)
If the operation is successful, this field contains the post-increment value of the requested counter document.

result.error (type: Object)
This field is populated if the operation failed. The contents of the error varies based on the type of error encountered, and commonly occurring fields are documented below.

result.error.code (type: number)
If present, the code of the SDK error that triggered this operation to fail. This is typically an internal numeric code.

result.error.name (type: string)
If present, the key is a stable token indicating the error that SDK encountered that caused this operation to fail. Error keys are stable over multiple releases and may be safely compared.

result.error.desc (type: string)
If present, a human readable description of the error that occurred. The description is for diagnostics and logging purposes only and may change over time. No programmatic logic should be tied to specific contents from this field.

exceptions
This API indicates errors via the error object in the return value. Exceptions are thrown only during system failure conditions.

Advanced DECREMENT:

results = couchbase.decrement(binding, meta)

This operation atomically decrements the field "*count*" in the specified document. The document must have the below structure:

```
{"count": 23} // 23 is the current counter value
```

The *decrement* operation returns the post-decrement value.

If the specified counter document does not exist, one is created with `count` value as 0 and the structure noted above. And so, the first returned value will be -1.

JavaScript numbers are accurate only till 53-bits. Hence, using *get()* followed by an *upsert()* with CAS rather than *decrement()* is recommended when using counts that may go beyond $\pm 2^{53}$.

Due to limitations in KV engine API, this operation cannot currently manipulate full document counters.

binding
The name of the binding that references the target bucket. The binding must have access level of *"read/write"*.

meta (type: Object)
The positional parameter (denoted by "*meta*" in the prototype above) represents the metadata of the operation. At minimum, the document key must be specified in this object.

meta.id (type:string)
The key of the document to be used for the operation. This is a mandatory parameter and must be of JavaScript *string* type.

result – the return value (type: Object)
The return object indicates success/failure of the operation, and the metadata of the operation.

result.success (type: boolean)
This field indicates if the operation was successful or not. It is always present in the return object.

result.meta (type: Object)
This field is present only if the operation succeeded. It contains metadata about the counter that was decremented (or created and decremented).

result.meta.id (type: string)
The key of the document that was inserted by this operation.

result.doc.count (type: number)
If the operation is successful, this field contains the post-decrement value of the requested counter document.

result.error (type: Object)
This field is populated if the operation failed. The contents of the error varies based on the type of error encountered, and commonly occurring fields are documented below.

result.error.code (type: number)
If present, the code of the SDK error that triggered this operation to fail. This is typically an internal numeric code.

result.error.name (type: string)
If present, the key is a stable token indicating the error that SDK encountered that caused this operation to fail. Error keys are stable over multiple releases and may be safely compared.

result.error.desc (type: string)
If present, a human readable description of the error that occurred. The description is for diagnostics and logging purposes only and may change over time. No programmatic logic should be tied to specific contents from this field.

exceptions
This API indicates errors via the error object in the return value. Exceptions are thrown only during system failure conditions.

Logging
A utility function, *log()* has been introduced to the language, which allows handlers to log messages. Messages logged using these functions go into their own files in the @evening subdirectory of Couchbase Server’s data directory. These files are distinct from system log files.

Each handler has its own set of log files. The log files are rotated to ensure the file size is capped. By default, each log file can grow up to 40MB before being rotated. 10 such files are retained.

The *log()* 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);
}
```

This function does not throw exceptions.

N1QL Queries

Top level N1QL keywords, such as SELECT, UPDATE, INSERT, are available as keywords in handlers. Operations that return values such as SELECT are accessible through a returned Iterable handle. Query results are streamed in batches to the Iterable handle as the iteration progresses through the result set.

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;
    }
    results.close();               // End the query and free resources held
}
```

The call starts[3] the query and returns a JavaScript Iterable object representing the result set of the query. The query is streamed in batches as the iteration proceeds. The returned handle can be iterated using any standard JavaScript mechanism including *for...of* loops.

The returned handle must be closed using the `close()` method defined on it, which stops the underlying N1QL query and releases associated resources.

All three operations, i.e., the N1QL statement, iterating over the result set, and closing the Iterable handle can throw exceptions if unexpected error arises from the underlying N1QL query.

As N1QL is not syntactically part of the JavaScript language, the handler code is transpiled to identify valid N1QL statements which are then converted to a standard JavaScript function call that returns an Iterable object with addition of a `close()` method.

The `N1QL()` call[3] is documented below for reference purposes but should not used directly as doing so would bypass the various semantic and syntactic checks of the transpiler (notably: recursive mutation checks will no longer function, and the statement will need to manual escaping of all N1QL special sequences and keywords).

[3] Also see note in "Language Change History" section regarding older construct

```
handle = N1QL(statement, [params], [options])
```

statement

This is the identified N1QL statement. This will be passed to N1QL via SDK to run as a prepared statement. All referenced JS variables in the statement (using the *$var* notation) will be treated by N1QL as named parameters.

params

This can be either a JavaScript array (for positional parameters) or a JavaScript map. When the N1QL statement utilizes positional parameters (i.e., $1, $2 ...), then *params* is expected to be a JavaScript array corresponding to the values to be bound to these positional parameters. When the N1QL *statement* utilizes named parameters (i.e., $name), then *params* is expected to be a JavaScript map object providing the name-value pairs corresponding to the variables used by the N1QL statement. Positional and named value parameters cannot be mixed.

options

This is a JSON object having various query runtime options as keys. Currently, the following settings are recognized:

> "consistency"
>
> This controls the consistency level for the statement. Normally, this defaults to the consistency level specified in the overall handler settings but can be set on a per statement basis. The valid values are "*none*" and "*request*".

return value (handle)

The call returns a JavaScript Iterable object representing the result set of the query. The query is streamed in batches as the iteration proceeds. The returned handle can be iterated using any standard JavaScript mechanism including *for...of* loops.

> close() Method on *handle* object (return value)
>
> This releases the resources held by the N1QL query. If the query is still streaming results, the query is cancelled.

Exceptions Thrown

The `N1QL()` function throws an exception if the underlying N1QL query fails to parse or start executing. The returned Iterable handler throws an exception if the underlying N1QL query fails after starting. The `close()` method on the iterable handle can throw an exception if underlying N1QL query cancellation encounters an unexpected error.

Timers

Handlers 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[4] 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.

The call returns the reference string if timer was created successfully. If multiple timers are created with the same unique reference, old timers with the same unique reference are implicitly cancelled. If the reference parameter is set to JavaScript null value, a unique reference will be generated.

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.

return value

If the ***reference*** parameter was null, this call returns the generated unique reference. Otherwise, the passed in ***reference*** parameter is the return value.

[4] Also see note in "Language Change History" section regarding API change

Exceptions Thrown

The `createTimer()` function throws an exception if the timer creation fails for an unexpected reason, such as an error writing to the metadata bucket.

Cancelling a Timer

Timers can be cancelled as follows:

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

callback

This function that was scheduled to be called when the timer fires, as supplied to the `createTimer()` call that is now being cancelled.

reference

This is the reference that was either passed in to the `createTimer()` call, or generated and returned by the `createTimer()` call in response to a null value for the incoming reference parameter.

return value

A boolean value indicating if the specified timer could be cancelled successfully. A *false* return value typically indicates the timer never existed or had already fired prior to the cancellation request.

Exceptions Thrown

The `cancelTimer()` function throws an exception if the timer cancellation fails for an unexpected reason, such as an error writing to the metadata bucket. (Note that cancelling stale or non-existent timers will be treated as a no-op and will not throw an exception).

cURL

The `curl()` function[5] provides a way of interacting with external entities using HTTP:

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

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.

[5] Also see note in "Language Change History" section regarding API change

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.

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.

Ability to include/exclude cipher rules setup at Couchbase Server level is currently unimplemented.

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.

```
var request = {
  path: '/person',
  params: {
    'action': 'create',
    'id': 23012
  },
  body: {
    'name': 'John Smith',
    'age': 25,
    'state': 'CA',
    'country': 'US',
  }
};

var response = curl('POST', profile_svc_binding, request);
if (response.status == 200) {
  var profile_id = response.body.profile_id;
  log("Successfully created profile " + profile_id);
}

```

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.

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

Terminology

Handler
A handler is a collection of JavaScript functions that together react to a class of events. 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 handler is captured in the below external elements, 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 handler

Deduplication
Couchbase does not store every version of a document permanently. Hence, when a Handler 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, Handlers 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 Handler manipulates documents in a bucket that serves as the source of mutations to this or any other Handler, a write originated by a Handler will cause a mutation to be seen by itself or another handler. These are called potentially recursive mutations.

Backwards Compatibility
Eventing project aims to retain language backwards compatibility in language constructs and minimize changes needed to handlers when upgrading through Couchbase Server versions.

Feature Stability Levels

"Committed" Language Constructs
All committed constructs will remain backwards compatible through all patch and minor releases, and at least one major release of Couchbase Server. We may change the semantics of a language construct in any given release but will ensure an older handler will continue to see the runtime behavior that existed at the time it was authored, until such behavior is deprecated and removed.

"Uncommitted" Language Constructs

Constructs may change and older behaviors will not be available in backwards compatibility mode once removed or changed. Using uncommitted features in production is discouraged.

"Internal" Language Constructs

These items are intended to illustrate working of the product and must not be used directly.

The *Language Version* Setting

Every handler records its desired language compatibility version in its settings section (visible in the UI under handler settings). This is a mandatory field. The UI selects the most current language version for newly created handlers.

Language Change History

	5.5	6.0	6.5	7.0
Feature: Bucket Operations	Committed	Committed	Committed	Committed
Feature: Timers	Uncommitted	Committed	Committed	Committed
Feature: Embedding N1QL	Uncommitted	Uncommitted	Committed	Committed
Feature: Curl	-	Uncommitted	Committed	Committed
Feature: Modifying docs in source bucket	-	-	Committed	Committed
Function: crc64()	-	-	Committed	Committed
Function: N1qlQuery	Internal	Internal	Removed	-
Function: N1QL()	-	-	Uncommitted	Uncommitted
Function: cancelTimer()	-	-	-	Committed
Argument: options in OnDelete()	-	-	-	Committed

Changes in 6.0.0:

Change to timer API

In versions prior to 6.0.0, there were two ways to create timers - ***`cronTimer()`*** and ***`docTimer()`***. Both these calls have been are removed in 6.0.0, and the functionality has been folded into the ***`createTimer()`*** call. As this was an Uncommitted feature in 6.0.x, backwards compatibility is not supported.

Changes in 6.5.0:

Change in behavior accessing non-existent items from a bucket

In versions prior to 6.5.0, the bucket Get operation would throw an exception when accessing missing objects. To be consistent with JavaScript, in 6.5.0 and later, accessing a missing key using bucket Get operation returning JavaScript *undefined* and does not throw an exception. As this was a Committed feature prior to 6.5.0, full backwards compatibility is made available using language versioning setting.

Replacing curl() with Committed version

The Uncommitted version of ***`curl()`*** available in 6.0.x has been replaced with the final Committed version from 6.5.0. As this was an Uncommitted feature in 6.0.x, backwards compatibility is not supported.

Change in N1qlQuery() class

The internal class ***`N1qlQuery()`*** used in the transpiled language has been replaced with a new internal class ***`N1QL().`*** As this is an internal artifact of an Uncommitted feature in 6.0.x, backwards compatibility is not supported.

Change in point of time when SELECT statements run

Prior to 6.5.0, N1QL SELECT statements would run only when iterator was first accessed and not when the SELECT was issued. This has changed. SELECT statements will start running as soon as issued. As N1QL features were Uncommitted prior to 6.5.0, backwards compatibility is not supported.

Changes in 7.0.0:

- A new argument named *options* has been added to OnDelete() method. This parameter indicates additional information about the delete event. Currently, it contains a single boolean entry '*expired'* that indicates if the deletion was due to document expiration. As the argument is an additional argument and JavaScript allows functions to ignore additional arguments, it is backwards compatible with prior versions' signature of the OnDelete() handler.

- New function cancelTimer() has been added to enable canceling pending timers.