Eventing 5.1 Specification

Operations

The following concepts shall be exposed to customers through UI with below semantics. Admin operations are accessible through UI, and later in 5.1 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 v8 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.

Deploy – DCP semantics

Deployment can be effected on DCP provider in three variations:

Deploy from Start

This choice currently affects only DCP observer. This choice causes DCP stream to start from sequence number 0. In other words, DCP data observer will cause function to visit each document at least once if deployed using this method.

Deploy from Now

This choice currently affects only DCP observer. This choice causes functions to start observing mutations from current sequence number of master of each vBucket. In other words, this will cause functions to visit documents modified after it is deployed.

Deploy from Prior Deployment

This choice currently affects only DCP observer. This option runs the point where the prior function left off. In other words, when documents mutate during the time when the prior version of the function was undeployed, and new version is deployed will be visited by the new function.

Deploy – Timer semantics

Deployment can be effected by Timer provider in two variants:

Retain timers

This choice causes all queued timers from prior versions of the function to fire on the latest version of the deployed function when it is activated. Note that the system will define an upper bound on maximum number of timers that can be accumulated in this manner.

Cleanup Timers

This choice causes all timers created by prior versions of the functions to be deleted, and only timers created after deployment of the current version of the function will be honored.

Pause

This operation will stop sending events to the function. Events that occur that are of interest to a paused handler will be queued up to system defined limits provided the event provider that originates such events supports replay semantics. Timers that fire will be temporarily deferred up to system defined limits.

Resume

This operation will restart sending events to the function. Events that had queued when the function was Paused will be sent. Deferred timers will now fire on the function.

Undeploy

This operation causes the function to stop processing events of all types, and shuts down the worker processes associated with the function. It releases any runtime resources acquired by the function. Functions in undeployed state allow code to be edited. An undeployed function retains memory of its prior deployment where necessary. Newly created handlers start in Undeployed state.

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 ephemeral 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 is not designed to be used on production environments. In addition, the 5.1 integration of v8 debugger is a DP feature, and does not provide correctness or functionality guarantees.

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. In 5.1, 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.

```
1.  function OnUpdate(doc, meta) {
2.    // Assuming 'dest' is a bucket alias binding
3.    var val = dest[meta.id];          // this is a bucket GET operation.
4.    dest[meta.id] = {"status":3};     // this is a bucket SET operation.
5.    delete dest[meta.id];             // this is a bucket DEL operation.
6.  }
```

N1QL

Top level N1QL keywords, such as SELECT, UPDATE, INSERT, are available as keywords in functions. Operations that return values 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. Any variable which is reachable from the scope of the N1QL query can be referred to using *$<variable>* syntax in the N1QL statement where parameters will be substituted according to the rules of named parameters substitution in the N1QL grammar specification.

The iterator we provide is an input iterator (elements are *read-only*). The keyword 'this' can not be used in the body of the iterator. The variables created inside the iterator are local to the iterator.

```
1.  function OnUpdate(doc, meta) {
2.      var strong = 70;
3.      var results =
4.         SELECT *                    // N1QL queries are embedded directly.
5.         FROM `beer-samples`         // Token escaping is standard N1QL style.
6.         WHERE abv > :strong;        // Local variable reference using : syntax.
7.      for (var beer of results) {    // Stream results using 'for' iterator.
8.          break;                     // Cancel streaming query by breaking out.
9.      }
10. }
```

Timers

Functions can register to observe wall clock time events. Such events can occur either standalone, or in reference to a specific document. 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 ordering between any two timers will be maintained. Timers only guarantee to fire at or after the specified time. Cron timers allow an opaque value to be provided, which is

made available to the callback when the timer fires. Opaque values are serialized and deserialized and hence are passed by value, and must be smaller than system defined limits.

Cron Timers

Cron timers allow a function handler to be called at a specific time. The opaque value stored when the timer is created forms the context for the timer callback. The time at which the callback function must be called must be specified as epoch time in seconds.

```
function checkProblems(ctx) {
    var res = SELECT * from inventory WHERE type = ctx.type;
    for (var item of res) {
       if (item.stock < 0) {
        log("Invalid stock status for " + item);
    }
}

function OnUpdate(item, meta) {
    if (item.stock < 0) {
        // 60 seconds from now, in epoch time (seconds)
        var time = Math.round((new Date()).getTime() / 1000) + 60;
        cronTimer(checkProblems, {"type": item.type}, time);
    }
}
```

Doc Timers

A doc timer is similar to a cron timer, except that the timer is associated with a document. Hence, doc timers follow the lifecycle of the document, including rollbacks. Doc timer callbacks receive the document key during timer creation. The time at which the callback function must be called must be specified as epoch time in seconds.

```
function monitorRefill(key) {
    var item = orders[key];
    if (item.stock == 0) {
        log("Possible refill problem for " + item)
    }
}

function OnUpdate(item, meta) {
    if (item.stock == 0) {
        // 24 hours from now, in epoch time (seconds)
        var time = Math.round((new Date()).getTime() / 1000) + 24 * 60 * 60;
        docTimer(monitorRefill, meta.id, time);
    }
}
```

Handler Signatures

The following event handlers are available in 5.1.

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.

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

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

Terminology

Binding

A binding is a construct that allows separating environment specific variables such as bucket names, external endpoint URLs, credentials etc. from the handler source code. It is primarily intended to enable functions to not require source changes during development to production workflows. It is recognized that this is not an absolute definition, and some parameters may be reasonably utilized as either a binding or a source code literal.

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.

Redeployment

Functions do not have a native concept of redeployment in 5.1. However, deployment with the DCP provider option of starting with last sequence number processed by the prior deployment of this function is sometimes referred to colloquially as redeployment.

State and Statelessness

These refer to the characteristic that any persistent state of a function is captured in entirety by the below, and any state that appears on the execution stack is ephemeral.

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