# Finite State Machine Component

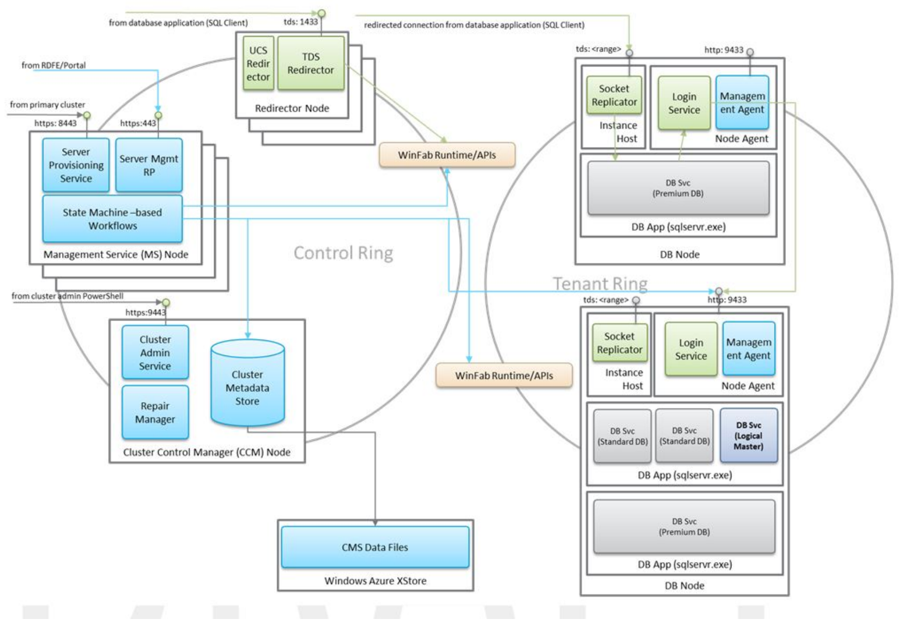
Last updated by | Vitor Tomaz | Aug 5, 2020 at 12:34 PM PDT

Contents

Sterling supports a workflow control model based on collaborating state machines, managed by the Finite State Machine (FSM) framework, that persist state to the Cluster Metadata Store (CMS).

The Finite State Machine (FSM) framework is a control framework for managing sequences of actions against entities based on their state. The framework implements a state machine-based approach that is optimized for managing a set of external resources that are represented by entities persisted to a SQL database.

The FSM framework will be used to manage actions on entities in a backing SQL Database store, controlling these actions based on entity state. While the framework is a generalized component capable of being used in many contexts, the focus is on management of Sterling cluster resources that are represented in a central (one per cluster) Cluster Metadata Store (CMS), held in a SQL database on the Control Ring.



Picture 1

This control model decomposes a high-level operation request at the REST API-level into one or more event-driven single-entity workflows that track changes typically to a single resource.

Entity workflows comprise one or more actions and/or state transitions. If an event triggering an entity workflow causes the entity to enter an unstable state, an automatically triggered action may cause further transitions, which may trigger further actions and further transitions. Actions may do work on the underlying resources and/or spawn events on other entities causing those entities to join the workflow.

SAWA v1 and Sterling management service expose the same APIs, which should allow a common data model representing operations to be used.

Ideally, management actions would update metadata in the store and affected resources in the cluster simultaneously. However, the resource providers are external to the store and may be unavailable. Actions may fail and need to be retried. Management actions are often composite, affecting multiple independent resources with updates taking different amounts of time, and potentially occurring in parallel. For example, provisioning a logical server instantiates a SQL application in WinFab, a database service under the application that will act as the logical master, a DNS record for the logical server and a WinFab alias for the database service. Each of these resources must be available and in a stable ‘ready’ state for the logical server to be fully available for use. The state of each resource is tracked in CMS by entities related directly or indirectly to the logical server entity.

The lifecycle of each entity type is described by a state machine with states, events, actions and transitions. Typically the state machine for an entity type that represents an external resource managed by some resource provider has stable states interspersed with transient states that represent work in progress. Actions can update a resource and its associated entity in the store and change the state of the entity recorded in the CCM store. An action on one entity can also raise events and trigger actions on other entities.

The notion of an unstable state, in which an action is automatically triggered, is supported. Workflows can be defined by chaining sequences of unstable states and actions. Persisting the state of an entity at every transition allows interrupted workflows to be restarted and will help ensure that resources don’t become stuck in some unexpected state in a workflow, and that the metadata and resources remain synchronized. Error states and recovery or undo actions can be defined to accommodate failures or unavailability of resources or resource providers.

At runtime the state machine framework operates over the metadata store and mediates actions on each entity (and its associated resource) based on the state of the entity in the store. At any point the overall state of all resources in the cluster can be inspected by querying the store.

Resources are managed through actions taken on entities in a metadata store that represent those resources. The entities in the metadata store are considered the 'source of truth' regarding the state of each resource; they are not a cache.

Requests can be processed synchronously or asynchronously. If a request is processed asynchronously the caller must return and inspect the entity state to determine the outcome. All requests to update an entity and its corresponding resource are accepted or rejected based on the state of that entity in the metadata store.

## State Machine

The FSM framework supports a state machine (SM) for each managed entity type. An executing state machine is associated with a single *managed entity* and manages update actions that influence that entity and any of its *subordinate entities*.

A state machine is defined by:

- The set of *events* that can be raised on the state machine.
- The set of *states*. An entity is always in exactly one state at any specific moment. One state is defined as the initial state. Each state identifies the events allowed in that state. States may be *stable* or *unstable*. Unstable states automatically execute an *action* that is mapped to the state on entry into that state.
- The set of legal *transitions* between states which may include transitions that start and end in the same state.
- The set of *actions* that may cause transitions between states;

## Managed Entity

An entity on which all CRUD actions are mediated by a state machine. Is implemented by a table in the underlying store.

The entity may have properties. Properties are publicly accessible (public get). But can only be privately set, either in a constructor or in an action.

A managed entity may have a relationship to one or more subordinate entities. A subordinate entity typically provides additional detail about the parent managed entity. A subordinate entity should be updated in the same transaction as its parent managed entity. If a subordinate entity has any meaningful state management requirement independent of its parent it should be promoted to a managed entity with its own state machine.

## State

A state implicitly constrains the values of properties and relationships of an entity and actions that can be invoked by restricting the events that are processed on it while the entity is in that state.

One state must be identified as the initial state in which the entity is first created.

One state can be identified as the deleted state in which the entity is unavailable. No transitions are allowed from the designated deleted state.

One or more states may be designated as terminal states, from which no transitions are allowed (e.g. to represent an archival or read-only states.)

Transitions between states are caused by actions.
A **stable state** is one in which transitions out of that state are caused only by an action triggered by an event, or an **unstable state** has exactly one action which will be automatically executed on entry into that state and which may cause a transition to some other state. An unstable state may also support events that trigger other actions. An event will be processed by an entity in an unstable state between retries of its automated action.

SQL constraints should be written by the developer in CMS to constrain values of properties and linked entities where these must be restricted based on the state. Violation of such constraints must prevent the state machine from committing an action that attempted to change to a state with inconsistent properties or links.

## Event

Events are raised on a state machine instance and are the primary mechanism by which a state machine can be influenced from the outside to trigger an action.

An event can be raised explicitly by an external caller or An event can be raised automatically on a periodic basis by a timer.

An event may be raised by an action in another state machine to extend a running workflow.

An event may be triggered by a subscription to a state being reached in another state machine or a timeout if the subscribed-to state(s) don’t occur within a configurable period of time.

An event is valid in one or more states.

An event can take parameters which are made available to the action triggered by the event.

Events are processed immediately and not queued or persisted.

Event validation and processing occur within a single transaction to ensure the entity does not change state between the state check and action.

Events can declare specific errors associated with invalid state which can be reported to the caller, thus a state machine could report not just that an event is invalid, but, for example, could indicate that the requested action is already in progress, or has already occurred.

## Action

Actions encapsulate all the work required to create update or delete entities and create, update and delete any associated resources.

Actions are implemented by developers as private methods that are executed by the FSM framework in response either to events being raised or the entity entering a state on which the action is mapped as the automatic action. Users of the FSM framework cannot invoke actions directly.

An action is mapped to one or more states in which it may be triggered, and can be mapped to an event.

An action is mapped to one or more target states indicating states to which it may transition the state machine and which must be enforced by the framework at runtime.In processing a state change or event, only a single action can be executed and will run to completion before any other event or state change is processed on the same entity. (This prevents interruption of the state machine at indeterminate points).

An action is atomic and executes within a transaction and can:

- Create or modify the resource instance that corresponds to the current entity.
- Update properties of the entity on which the triggering event was raised, or create, update or delete any subordinate entities
- Get the state of another entity including optionally taking a lock on that entity to prevent other actions from changing the entity.
- Raise ‘nested’ events on other entity state machines so as to trigger an action.

Create a new managed entity (instance) of any entity type, optionally passing constructor arguments.

- Transition the entity to a target state which may be the current state.

If an action raises an event synchronously on another entity it blocks the action pending the outcome of that event, which will occur when that entity enters a stable state.

If an action raises an event asynchronously on another entity the action will not block and may continue, and, for example, transition the state of the entity into a state indicating that other work is in progress. The same or later action may inspect the entity determine its state in order to determine the outcome of the earlier event.

An action may determine its outcome based on the response from a called task or raised event, for example an action may transition to a success state, an error state or back to the same state depending on the response from a an action on a resource provider. Transitioning to the same state will cause execution of any automatic action on that state, which can be used as a mechanism to retry an action. The action can programmatically manage the number of retries and the delay before re-execution on retry to manage the load on the system.

## Transition

A transition is considered to be a ‘movement’ of an entity into a specified state. While typically an entity may move or transition into a different state from its prior state, an entity can also transition into the same state. If the state is unstable, re-entering the same state will trigger the automatic action on the state (if there are no pending events).

A transition can occur only as a result of an action. The set of possible transitions an action can cause is declared as part of the definition of the action. If an action is terminated then the entity is considered to re-enter the state from which the action was invoked.

An action can determine within its implementation logic which transition to make from among its declared possible transitions. For example, if a resource is unavailable it may return to its current state to retry, if a resource is updated successfully it may transition to a success state while if an error occurs it may transition to an error state from a which another action is triggered to recover from the error.

From the standpoint of the logic in the action, the transition can occur at any point although the transition will only be visible to other actions or consumers of the store once the transaction in which the action is executed is committed. If the transaction is rolled back the transition will not occur and the entity will be considered to re-enter its current state (which may trigger an automatic action).

From the standpoint of the state machine a transition takes zero time (i.e. there is no intermediate 'transitioning' state).

A wait time can be declared on a transition into an unstable state which will defer execution of the automatic action.

## Workflow

Each workflow in the collection contains info about the workflow (start/end times etc.) and reflects a summary of the current state of the participant FSMs. The status of an individual workflow reflects a summary of the error statuses and errors encountered by all the state machines that participated in the workflow. At the API level, it will typically be important to simply return a success or failure response to the client.

An unstable state is defined as a state to which an action is mapped without an associated event. Only one such automated action is allowed per state. At runtime, when an entity enters into an unstable state the automated action will be executed. On completion, this automated action may transition the entity into an unstable state which will cause an action to be executed. This mechanism of chaining transitions from automated actions to unstable states which trigger further actions allows workflows to be defined.

Workflows are not defined explicitly or imperatively, but are an intentional by-product of the design of one or more state machines.

A workflow will result as soon as an entity enters an unstable state and will continue until a stable state is reached.

As the state of each entity is persisted to the store, workflows are restart-able in the event of a failure of any of running state machines.

To ensure predictability and restart-ability actions should be idempotent.

Consideration should be given to designing state machines such that if an error occurs the system transitions into an unstable error state which triggers appropriate recovery or undo actions.

Entities involved in the same workflow instance can be identified through the use of a request Id. Note that depending on the state machine designs, this request Id cannot be relied on across state changes if the state machines allow interruption by other workflows with a different request Id.

## Subscription Event

An event on a state machine in a stable state starts a chain of one or more actions (a workflow) which may advance the entity through a series of state changes. Any action in that workflow may create or raise events on other state machines and join those state machines into the workflow. Each state machine that joins a workflow is considered a successor to the state machine that created it or raised the event on it causing it to join the workflow.

An event on a state machine can subscribe to a specified state or pattern of states among its successor state machines. When a qualifying pattern is detected the subscription event can be raised. Like other events, a state machine must declare in what states a subscription event is valid. The event will be raised on the state machine in one of two cases:

The subscribing state machine is in a state where a subscription event is valid and a state change occurs among its successors which causes the successor pattern to match the subscription.

The subscribing state machine enters a state in which a subscription event is valid and the successor pattern matches its subscription.

## Supportability

The framework is resilient to failure of an instance of a running state machine such that another instance may, where required by the state machine, pick up and continue a workflow started by the failed instance. The framework supports and makes monitoring easier, including emitting telemetry used for tracking workflow execution progress and measurement of key performance indicators (KPIs).

XEvents are emitted from throughout the Management Service stack, starting at the API layer and descending through the workflow and state machine management (FSM) into actions and at the CMS database level. Request Identifier is passed down through the stack and is included in all XEvents , allowing correlation of the events tracking every step in the execution of the operation. These events are gathered in MDS where the can be used to troubleshoot individual operations executions in detail.

To support reliability and performance analysis in the aggregate, information from these XEvents is rolled up and represented in the data warehouse to allow more efficient queries to be executed to provide KPIs and track trends and provide other useful insights.

Three levels of detail are provided that represent progressive drill-down into the detail of operations:

- Operations, allowing analysis of overall operation execution, duration, success and failure rates, and overall failure causes. Operations are tracked from the point the operation commences to the operation completes. Operations subsume the representation of workflows which operate at same level of abstraction.
- Events, allowing similar analysis of the behavior of individual state machines in response to specific events. Events are tracked in the aggregate from the point the event is raised or triggered on a state machines to the point that state machine settles in a stable state)
- Transitions, allows similar analysis of specific steps in a workflow, allowing analysis of the outcomes of specific actions. Transactions are tracked from a start state to a subsequent state (which may be the same as the start state), identifying where the transition was triggered by an event or by an action, where the end state is an error state or an action raised an error.

## Database Model

### tbl_operations

One row per operation executed via the management service API.

Note that an operation refers to the logical unit of work triggered by an API request. In some cases, where an async interaction pattern is used, multiple API calls may be made, first to initiate and then to track progress of an operation that is executed asynchronously WRT the original request.

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| **PK** | operation_id | integer | Surrogate key, issued in start time sequence | |
| | request_id | guid | The request id used throughout the ensuing workflow. Used to correlate with events and transitions. | |
| | Name | text | Name of the operation | |
| | workflow_start_time | time | The time the operation was received. | |
| | workflow_end_time | time | The time the operation was completed. | |
| | operation_first_response_time | time | The time the operation first returned a response to the caller. May be well before the workflow complete time for a long-running workflow which uses an async response pattern. | |
| | Outcome | ~~Bool~~ { canceled\| failed \| hung \| success } | True, if no participating state machines reported an error. Error might be reported by an action or that a state machine is in an error state. | |
| | error_id | [...] | The error reported by the operation to the caller. This error might summarize one or more errors that occurred in the execution of the operation. | Y |
| | restart_count | integer | The number of times the workflow was restarted due to FSM failure during its execution. | |
| | data_cluster_id | [...] | Reference to the data cluster (SQL Cluster) on which the operation was executed. | Y |

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| | logical_server_id | [...] | Reference to the logical server on which the operation executed or which contained the database on which the operation was executed. | Y |
| | logical_database_id | [...] | For a database operation, a reference to the logical database on which the operation was executed. | Y |
| | target_data_cluster | [...] | For a cross-SQL cluster operation, reference to target data cluster (SQL Cluster). | Y |
| | target_logical_server | [...] | For a cross-sever operation, the name of the target server. | Y |
| | target_logical database_id | [...] | For a cross-database operation, the id of the target database. | Y |

### tbl_state_machine_events

One row per event raised or triggered on a state machine within the context of a workflow. Tracks the event from initial triggering to the state machine returning to a stable state. Includes events raised programmatically, by subscription or by timer.

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| PK | operation_id | integer | Identifier of the operation within which this state machine event was raised. | |
| PK | event_id | integer | Identifier of the event within the operation. Issued in start time sequence | |
| | request_id | guid | Request Id of the originating operation | |
| | workflow_position | text | Workflow position of the subject state machine when the event occurred. | |
| | state_machine_type | text | The type of state machine on which the event was raised or triggered. | |
| | Event | text | The name of the event that was raised or triggered. | |
| | Keys | text | The state machine instance identifier. Serialized form of the key field values in key ordinal order | |
| | logical_server_id | [...] | For server-scoped operations, the reference to the logical server. | |
| | event_type | {raised\|subscription\|timer} | The type of event. | |
| | start_state | text | The state of the state machine in which the event was raised. | |
| | end_state | text | The state of the state machine at the conclusion of the processing of the event and any subsequent actions on this state machine triggered by the event. | |
| | start_time | time | The time that the event processing started. | |
| | end_time | time | The time that the event processing ended. | |

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| | error_end_state | bool | True if the end state is an error state | |
| | error_reported | bool | True if an error was reported by one or more actions invoked by this event. | |
| | Error | [...] | The last error reported by any action during processing of the event. | Y |
| | Exception | bool | True if the event failed with an exception | |
| | exception_details | text | The exception returned by the event | |
| | restart_count | integer | The number of times the state machine was restarted during execution of the event. Normally 0. | |
| | data_cluster_id | [...] | Reference to the data cluster (SQL Cluster) on which the operation was executed. | Y |
| | logical_server_id | [...] | Reference to logical server on which the operation executed or which contained the database on which the operation was executed. | Y |
| | logical_database_id | [...] | For a database operation, the reference to id of the database on which the operation was executed. | |
| | target_sql_cluster | [...] | For a cross-SQL cluster operation, the name of the target SQL cluster. | Y |
| | target_logical_server_id | [...] | For a cross-server operation, the name of the target server. | Y |
| | target_logical database_id | [...] | For a cross-database operation, the id of the target database. | |

## tbl_state_machine_transitions

One row per state-state transition occurring within the context of a State Machine Event. This table tracks a transition from the event that triggers to the state machine returning to a stable end state. The end state may be the same as the start state. The end state may be reached in a single step or multiple steps. Simple state transitions should be included for completeness

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| PK | operation_id | integer | The operation context in which this transition occurred | |
| PK | event_id | integer | The event context in which this transition occurred | |
| PK | transition_ordinal | integer | The sequence number of this transition within the event | |
| | transition_type | {Event \| EventAction \| AutoAction} | Indicates whether the transition occurred as a result of an event, was caused by an event-triggered action or was caused by an auto-executed action. | |
| | action_name | text | The name of the action causing the transition. More than one action may cause the same transition. | |
| | start_state | text | The state of the state machine in which the event was raised. | |
| | start_state_health | {normal \| error} | Indicates if the start state was a normal state or an error state | |
| | start_state_entry_time | time | The time the state machine entered the start state. | |
| | transition_start_time | time | The time that the transitioning started – either the time that the event was received or the eventual transitioning action started. For an auto action this is the start of the action occurrence that actually transitioned the state which may have been substantially after the state was entered, either because of a programmed delay or because of a retry loop. | |
| | retry_count | integer | The number of times an auto action was executed before transitioning the state machine to a different state. | |

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| | end_state | text | The state of the state machine at the conclusion of the processing of the event and any subsequent actions on this state machine triggered by the event. | |
| | end_state_health | {normal \| error} | Indicates if the end state was a normal state or an error state | |
| | transition_time | time | The time that the entity transitioned to the transition end state | |
| | action_start_time | time | The time that the action that transitioned the state machine started. | |
| | Outcome | text | For an action-driven transition, the outcome as reported by the action. | |
| | error_id | […] | The error reported by any action during the transition. | Y |

### tbl_workflow_exceptions

One row per exception occurring within the context of a State Machine Event.

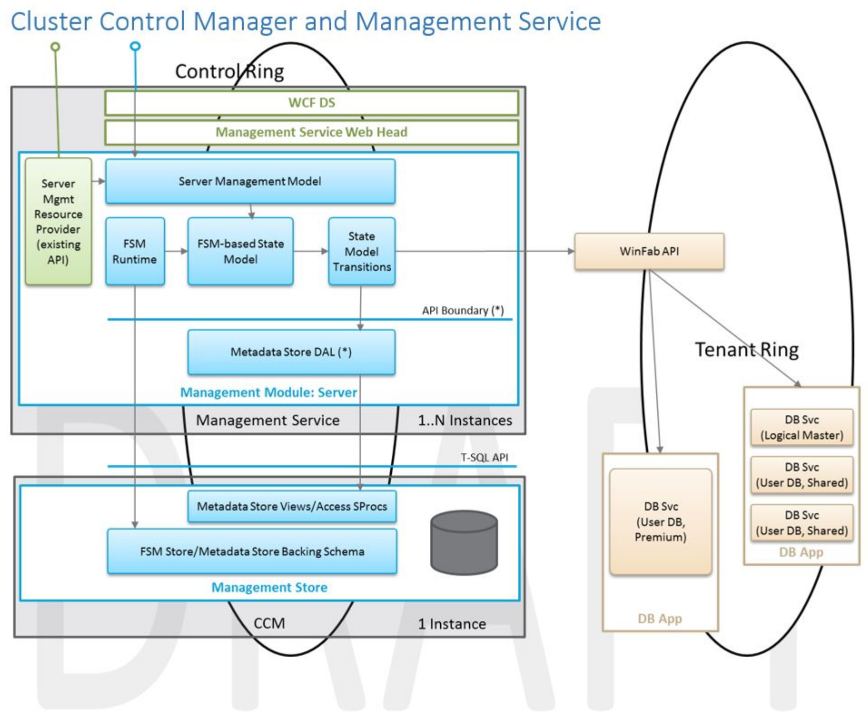
| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| PK | exception_id | integer | The operation context in which this transition occurred | |
| PK | Exception | text | Exception details | |

### tbl_workflow_errors

One row per error.

| | Column | Data Type | Description | Dim |
|---|---|---|---|---|
| PK | error_id | integer | The id of the error. | |
| | error_details | text | Error description. | |
| | error_kind | {enum} | The type of error. | |

## Cluster Control Manager and Management Service



Picture 2

### Cluster Control Manager (CCM)

The Cluster Control Manager (CCM) node is hosting a SQL server, set up as an SQL Azure instance on a remote storage. The CCM instance is hosting a database containing Metadata Store (CMS). There are conceptually two layers there:

- Metadata Store backing schema – the actual relational schema describing the state of the cluster. Information it contains will be, among others:
  - State and properties of logical servers on the cluster
  - Mapping between logical servers and WinFab application instances and service instances.
  - Mapping between logical servers and Windows Azure storage accounts (as needed)
  - State and properties of logical databases and their containment in the logical servers. This schema is also a direct backing store for the finite state machines. Metadata in CMS is considered the source of truth for the cluster. The store is not a cache. It must neither be updated optimistically before resource updates nor after the fact. At an individual resource level, the state of an entity must indicate whether the resource it represents is in a stable state or is subject to a workflow acting to change its state.
- Metadata Store Views and Access API – a set of T-SQL views and stored procedures that hides the actual schema and allows for it to evolve. We will ensure through the right permission system that no service has direct access to the Metadata Store schema.

### Management Service

Management Service is one of the services running on the Control Ring. Management Service will be configured to run as N independent, stateless instances. The number of instances will be (eventually) dynamically adjusted based on the workload.

Management Service is providing a common web service head infrastructure that allows us to expose standardized ODATA web service using WCF Data Services framework (part of .NET). The actual model that defines and implements the API is provided through a Management Module. For the functionality we are building as part of this project we will be defining a new Server Management Module, which is a replacement for an existing one of the same name. The module will claim a new API namespace to not conflict with the existing code and APIs.

In the management module we call out the following layers :

- Metadata Store DAL (Data Access Layer) – a simple set code that allows higher layers to call the T-SQL APIs without constructing T-SQL requests every time. This layer should be defined as a reusable library that can be provided to other components as well.

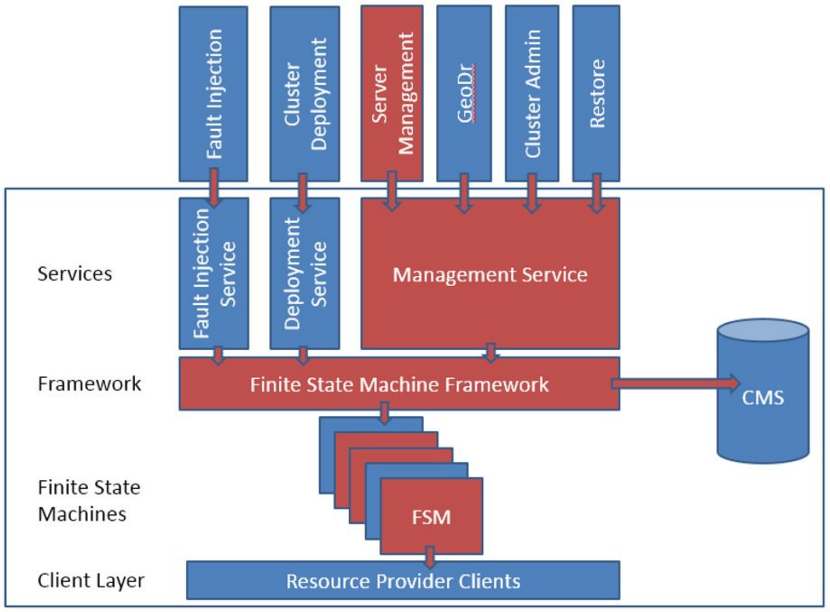
- State Model Transitions – a layer that codifies the behavior of the management operations. By separating the behavior from the state we can test it in isolation and we can update/version/substitute it with different behaviors easily.

- Finite State Machine (FSM) – this is a core layer of the provisioning workflows. It consists of the FSM-based model for each set of entities and an FSM runtime that holds the in-memory state of the entities and runs the necessary transitions.

  The FSM runtime is a stand-alone and reusable component. The FSM state machine is a set of classes, attributed with FSM tags that describe the state machines, with possible states and transitions. The transitions are simply invoking the State Model Transitions code. The state itself is persisted and loaded back through the Management Store Object Model.

- Server Management Model – is the top layer that defines the ODATA model. It is implemented as a set of classes, combined through collections and navigational properties into a set of EDM/ODATA entities that get surfaced as web API through WCF Data Services layer. The sole purpose of this layer is to define the shape of the API (the URIs) and the shape of responses for each request. It also implements any custom ODATA action verbs.

- Server Management Resource Provider – this is an REST API supports the UI Portal and Windows Azure PowerShell. The implementation of the API will call into the Server Management Model, or into the FSM State Model directly.

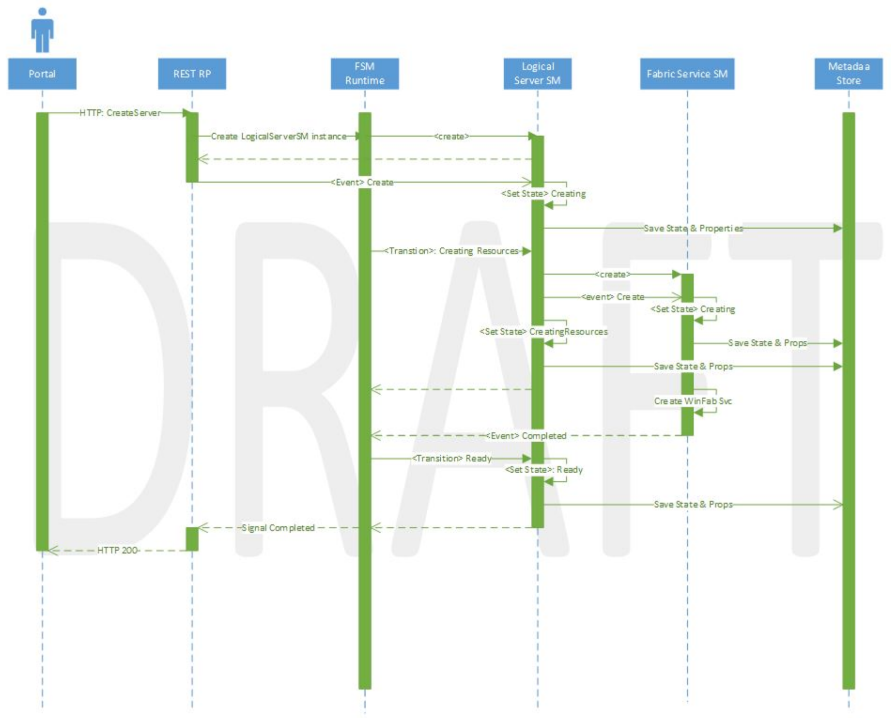


Picture 3

## Workflow of creating a logical server:

A logical server provisioning requires multiple resources are created and initialized and are available in a stable state before the user can create databases on the server. To accomplish provisioning a logical server, a new logical server entity is created in the CCM store. The logical server entity is transitioned into a Creating state which will invoke the action to start the provisioning of the resources associated with the logical server. A SQL application (corresponding to a SQL instance) is created in WinFab and then a database service is created on this SQL instance and then configured as the logical master database for the logical server. An alias is created to map the database to the WinFab service. In parallel a DNS record is created asynchronously. When all resource provisioning is completed the logical server instance transitions to a ready state and is available for use.

Transitions from non-existing server to a one in Ready state.



Picture 4

Description:

- Portal makes a HTTPs call to the Resource Provider, to create a logical server.
- Resource Provider runs in process with Finite State Machine (FSM) Runtime.
- Resource Provider creates a new instance of Logical Server State Machine (SM).
- Resource Provider sends a Create event to the new SM
- Logical Server SM transitions to Creating state as a result of the event. The state is saved into the Metadata Store.
- FSM framework detects that there is a SM in a Transient State and initiates transition to the next one – Creating Resources.
- The transition starts one or more additional state machines, each of them is signaled with an event and saved to the Metadata Store.
- The transition completes and the Logical Server SM is now in Stable State – CreatingResources.
- The newly spawn SMs are progressing. As part of their transitions, external resources are provisioned.
- Eventually the other SMs complete and signal back (through an event) that they are done.
- The new event triggers the transition for Logical Server SM to Ready state.
- The FSM Runtime signals back the Resource Provider that the entire workflow has completed.
- The Resource Provider returns HTTP 200 to the caller. Note also that the RP could have returned HTTP 202 earlier in the call and the portal can call back to check the progress of a long-running operation.

**How good have you found this content?**