Document Identifier: DSP2050

Date: 2017-06-30

Version: 1.0.0

Redfish Composability White Paper

Supersedes: None

Document Class: Informative

Document Status: Published

Document Language: en-US

CONTENTS

Foreword

The Redfish Composability White Paper was prepared by the Scalable Platforms Management Forum of the DMTF.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. For information about the DMTF, see http://www.dmtf.org.

Acknowledgments

The DMTF acknowledges the following individuals for their contributions to this document:

- Rafiq Ahamed K - Hewlett Packard Enterprise
- Jeff Autor - Hewlett Packard Enterprise
- Michael Du - Lenovo
- Jeff Hilland - Hewlett Packard Enterprise
- John Leung - Intel Corporation
- Steve Lyle - Hewlett Packard Enterprise
- Michael Raineri - Dell Inc.
- Paul von Behren - Intel Corporation

Introduction

As the world is transitioning to a software-defined paradigm, there is a need for hardware management capabilities to evolve to address that shift in the data center. In the context of disaggregated hardware, management software needs the ability to conjoin the independent pieces of hardware, such as trays, modules, silicon, etc., together to create a composed logical system. These logical systems function just like traditional industry standard rackmount systems. This allows users to dynamically configure their hardware to meet the needs of their workloads. In addition, users are able to manage the life cycle of their systems, such as adding more compute to their logical system, without having to physically move any equipment.

Redfish is an evolving hardware management standard that is designed to be flexible, extensible, and interoperable. Redfish contains a data model that is used to describe composable hardware, as well as an interface for clients to manage their compositions. This document helps implementers and clients understand the Redfish Composability data model as well as how composition requests are expected to be formed.

Modeling for Composability

If a Redfish service supports Composability, the Service Root resource will contain the `CompositionService` property. Within the Composition Service, a client will find the inventory of all components that can be composed into new things (Resource Blocks), descriptors containing the binding restrictions of the different components (Resource Zones), and annotations informing the client as to how to form composition requests (Collection Capabilities). The following sections detail how these things are reported by a Redfish service.

1. Composition Service

The Composition Service is the top level resource for all things related to Composability. It contains status and control indicator properties such as `Status` and `ServiceEnabled`. These are common properties found on various Redfish service instances. It also contains links to its collections of Resource Blocks and Resource Zones through the properties `ResourceBlocks` and `ResourceZones` respectively. Resource Blocks are described in the Resource Blocks section, and Resource Zones are described in the Resource Zones section.

Example Composition Service Resource:

```
{
    "@odata.context": "/redfish/v1/$metadata#CompositionService.CompositionService",
    "@odata.type": "#CompositionService.v1_0_0.CompositionService",
    "@odata.id": "/redfish/v1/CompositionService",
    "Id": "CompositionService",
    "Name": "Composition Service",
    "Status": {
        "State": "Enabled",
        "Health": "OK"
    },
    "ServiceEnabled": true,
    "ResourceBlocks": {
        "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks"
    },
    "ResourceZones": {
        "@odata.id": "/redfish/v1/CompositionService/ResourceZones"
    }
}
```

2. Resource Blocks

Resource Blocks are the lowest level building blocks for composition requests. Resource Blocks contain status and control information about the Resource Block instance. They also contain the list of components found within the Resource Block instance. For example, if a Resource Block contains one Processor and four DIMMs, all of those components will be part of the same composition request, even if only one of them is needed. In a completely disaggregated system, a client would likely find one component instance within each Resource Block. Resource Blocks, and their components, are not in a state where system software is able to use them until they belong in a composition. For example, if a Resource Block contains a Drive instance, the Drive will not belong to any given Computer System until a composition request is made that makes use of its Resource Block.

The property `ResourceBlockType` contains classification information about the types of components found on the Resource Block that can be used to help clients quickly identify a Resource Block. Each `ResourceBlockType` is associated with specific schema elements that will be contained within that Resource Block. For example, if the value `Storage` was found in this property, a client would know that this particular Resource Block contains storage related devices, such as storage controllers or drives, without having to drill into the individual component resources. The value `Compute` has special meaning; this is used to describe Resource Blocks that have bound processor and memory components that operate together as a compute subsystem.

The property `CompositionStatus` is an object that contains two properties: `CompositionState` and `Reserved`. `CompositionState` is used to inform the client of the state of this Resource Block regarding its use in a composition. `Reserved` is a writeable flag that clients can use to help convey that this Resource Block has been identified by a client, and that the client will be using it for a composition. If a second client that is attempting to identify resources for a composition sees the `Reserved` flag set to true, the second client should consider it allocated and not use it; the second client should move on to the next Resource Block for further processing. The Redfish service does not provide any sort of protection with the `Reserved` flag; any client can change its state and it is up to clients to behave fairly.

There are several arrays of links to various component types, such as the `Processors`, `Memory`, and `Storage` arrays. These links ultimately go to the individual components that are within the Resource Block. These components are made available to the new composition after a composition request is made. The `ComputerSystems` array is used when a Resource Block contains one or more whole Computer Systems. This gives the client the ablity to create a single composed Computer System from a set of smaller Computer Systems.

The `Links` property contains references to related resources. The `Chassis` array contains the Chassis instances that contain the resources within the Resource Block. The `ComputerSystems` array contains the Computer System instances that are consuming the Resource Block as part of a composition. The `Zones` array contains links to the Resource Zones that contain the Resource Block.

Example Resource Block Resource:

```
{
    "@odata.context": "/redfish/v1/$metadata#ResourceBlock.ResourceBlock",
    "@odata.type": "#ResourceBlock.v1_0_0.ResourceBlock",
    "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/DriveBlock3",
    "Id": "DriveBlock3",
    "Name": "Drive Block 3",
    "ResourceBlockType": [ "Storage" ],
    "Status": {
        "State": "Enabled",
        "Health": "OK"
    },
    "CompositionStatus": {
        "Reserved": false,
        "CompositionState": "Composed"
    },
    "Processors": [],
    "Memory": [],
    "Storage": [
        {
            "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/DriveBlock3/
Storage/Block3NVMe"
        }
    ],
    "Links": {
        "ComputerSystems": [
            {
                "@odata.id": "/redfish/v1/Systems/ComposedSystem"
            }
        ],
        "Chassis": [
            {
                "@odata.id": "/redfish/v1/Chassis/ComposableModule3"
            }
        ],
        "Zones": [
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceZones/1"
            },
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceZones/2"
            }
        ]
    }
}
```

In the above example, the Resource Block is of type `Storage`, and it contains a single storage entity. From the `CompositionStatus`, it is noted that the Resource Block is currently used in a composition, and in the `Links` section, it is being used by the Computer System `ComposedSystem`.

3. Resource Zones

Resource Zones describe to the client the different composition restrictions of the Resource Blocks reported by the service; Resource Blocks that are reported in the same Resource Zone are allowed to be composed together. This prevents clients from having to perform try-and-fail logic to figure out the different restrictions that are in place for a given implementation. In addition, each Resource Zone leverages the Collection Capabilities annotation to describe what each Resource Zone is able to compose. This is described in more detail in the Collection Capabilities section.

Example Resource Zone Resource:

```
{
    "@odata.context": "/redfish/v1/$metadata#Zone.Zone",
    "@odata.type": "#Zone.v1_1_0.Zone",
    "@odata.id": "/redfish/v1/CompositionService/ResourceZones/1",
    "Id": "1",
    "Name": "Resource Zone 1",
    "Status": {
        "State": "Enabled",
        "Health": "OK"
    },
    "Links": {
        "ResourceBlocks": [
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
ComputeBlock1"
            },
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock3"
            },
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock4"
            },
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock5"
            },
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock6"
            },
            {
                "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock7"
            }
        ]
    },
    "@Redfish.CollectionCapabilities": {
        "@odata.type": "#CollectionCapabilities.v1_0_0.CollectionCapabilities",
        "Capabilities": [
            {
                "CapabilitiesObject": {
                    "@odata.id": "/redfish/v1/Systems/Capabilities"
                },
                "UseCase": "ComputerSystemComposition",
                "Links": {
                    "TargetCollection": {
                        "@odata.id": "/redfish/v1/Systems"
                    }
                }
            }
        ]
    }
}
```

In the above example, the Resource Blocks `ComputeBlock1`, `DriveBlock3`, `DriveBlock4`, `DriveBlock5`, `DriveBlock6`, and `DriveBlock7` are all in the same Resource Zone. In addition, the Collection Capabilities for the Resource Zone shows that this Resource Zone is capable of producing Computer Systems for the collection `/redfish/v1/Systems`.

4. Collection Capabilities

Collection Capabilities will be found on Resource Zones and on the Resource Collections themselves. This is because Collection Capabilities can be applied to things outside of the context of Composability. Collection Capabilities can be identified by the `@Redfish.CollectionCapabilities` annotation in the response body. This annotation is used to inform the client how to form the request body for a create (POST) operation to a given collection based on a specified Use Case, which will result in a new member being added to the given collection.

4.1. Collection Capabilities Annotation

Within the Collection Capabilities annotation, there is a single property called `Capabilities`. This is an array to identify all of the capabilities for a given Resource Zone or Resource Collection. Inside each instance of the `Capabilities` array is an object to describe a particular capability.

The `CapabilitiesObject` property contains a URI to the underlying object instance that describes the payload format. This is described further in the next section.

The `UseCase` property is used to inform the client of the context of a particular create (POST) operation. The table below shows the different values for `UseCase` as used by Composability. Each value corresponds with a specific type of resource being composed in addition to a type of composition for the request.

`UseCase` Value	Composed Resource	Type of Composition
`ComputerSystemComposition`	`ComputerSystem`	Specific

The `TargetCollection` property inside the `Links` object contains the URI of the Resource Collection that accepts the given capability. A client will be able to perform a create (POST) operation against this URI as described by the contents of the `CapabilitiesObject`.

Example Collection Capabilities Annotation:

```
{
    "@Redfish.CollectionCapabilities": {
        "@odata.type": "#CollectionCapabilities.v1_0_0.CollectionCapabilities",
        "Capabilities": [
            {
                "CapabilitiesObject": {
                    "@odata.id": "/redfish/v1/Systems/Capabilities"
                },
                "UseCase": "ComputerSystemComposition",
                "Links": {
                    "TargetCollection": {
                        "@odata.id": "/redfish/v1/Systems"
                    }
                }
            }
        ]
    },
    ...
}
```

The above annotation contains a single capability. From the `UseCase`, this capability describes how to form a create (POST) request to create a new Computer System from a set of specific Resource Blocks. In addition, the `TargetCollection` property indicates that a client can make the request to the Resource Collection `/redfish/v1/Systems`; new instances of the resource created by the client will be found in that collection.

4.2. Collection Capabilities Object

The Collection Capabilities Object follows the schema of the new resource a client is able to create. For example, if the object is describing how to form a request to create a new Computer System instance, the object's type will be `ComputerSystem.vX_Y_Z.ComputerSystem`, where `vX_Y_Z` is the version of `ComputerSystem` supported by the service.

The object itself contains annotated properties the client can use in the body of the create (POST) operation. It also lists optional properties, and any restrictions properties may have after the new resource is created. The table below describes the different annotations used on the properties within the Collection Capabilities Object.

Property Annotation	Description
`Redfish.RequiredOnCreate`	The client must provide the given property in the body of the create (POST) request.
`Redfish.OptionalOnCreate`	The client may provide the property in the body of the create (POST) request.
`Redfish.SetOnlyOnCreate`	If the client has a specific value needed for the property, it must be provided in the body of the create (POST) request; this property is likely a "Read Only" property after the resource's creation.
`Redfish.UpdatableAfterCreate`	The client is allowed to update the property after the resource is created.
`Redfish.AllowableValues`	The client is allowed to use any of the specified values in the body of the create (POST) request for the given property.

Example Collection Capabilities Object:

```
{
    "@odata.context": "/redfish/v1/$metadata#ComputerSystem.ComputerSystem",
    "@odata.type": "#ComputerSystem.v1_4_0.ComputerSystem",
    "@odata.id": "/redfish/v1/Systems/Capabilities",
    "Id": "Capabilities",
    "Name": "Capabilities for the Zone",
    "Name@Redfish.RequiredOnCreate": true,
    "Name@Redfish.SetOnlyOnCreate": true,
    "Description@Redfish.OptionalOnCreate": true,
    "Description@Redfish.SetOnlyOnCreate": true,
    "HostName@Redfish.OptionalOnCreate": true,
    "HostName@Redfish.UpdatableAfterCreate": true,
    "Boot@Redfish.OptionalOnCreate": true,
    "Boot": {
        "BootSourceOverrideEnabled@Redfish.OptionalOnCreate": true,
        "BootSourceOverrideEnabled@Redfish.UpdatableAfterCreate": true,
        "BootSourceOverrideTarget@Redfish.OptionalOnCreate": true,
        "BootSourceOverrideTarget@Redfish.UpdatableAfterCreate": true,
        "BootSourceOverrideTarget@Redfish.AllowableValues": [
            "None",
            "Pxe",
            "Usb",
            "Hdd"
        ]
    },
    "Links@Redfish.RequiredOnCreate": true,
    "Links": {
        "ResourceBlocks@Redfish.RequiredOnCreate": true,
        "ResourceBlocks@Redfish.UpdatableAfterCreate": true
    }
}
```

In the above example, three properties are marked with the `Redfish.RequiredOnCreate` annotation: `Name`, `Links`, and `ResourceBlocks` inside of `Links`. All other properties are annotated with `Redfish.OptionalOnCreate`. However, both `Name` and `Description` are annotated with `Redfish.SetOnlyOnCreate`, meaning they cannot be modified after the new resource is created.

Types of Compositions

The Redfish Composability data model provides flexibility for service implementers to report different Composition Types based on their needs. The service informs the client of the type of composition request based on the `UseCase` property found in the Collection Capabilities Annotation. The existing Redfish Composability model has defined one type called Specific Composition.

5. Specific Composition

The Specific Composition allows clients to create and manage the life cycle of composed resources through predefined Resource Blocks and Resource Zones. Because Resource Blocks are self-contained entities within a Resource Zone, clients are able to pick and choose specific Resource Blocks for their composition request.

An example of choosing a Resource Block according to the binding rules and providing details of specific Resource Blocks in the a create (POST) request can be found in the Create a Composed Resource section.

Another industry standard server design that fits into the example of Specific Composition is defined in the Bladed Partitions Mockup. In this example, a Multi-Blade Enclosure consisting of a disaggregated hardware chassis can be bound together to create what are called partitioned servers. These partitions can be composed using the Specific Composition. The Redfish service implements each blade within the enclosure as a Resource Block with `ResourceBlockType` set to either `Compute` or `Storage`, and allows the clients to combine multiple Resource Blocks to create a composed Computer System, which is a partitioned server.

Example Create (POST) Body for a Specific Composition:

```
{
    "Name": "Sample Composed System",
    "Links": {
        "ResourceBlocks": [
            { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
BladeComputeBlock1" },
            { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
BladeComputeBlock5" },
            { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
BladeStorageBlock8" }
        ]
    }
}
```

Appendix

6. General Workflow for a Client

Here are the few operations that a client is expected to use during creation and management of Composed Systems using the Redfish Composition models. The examples below expect the client will have a valid Redfish session or Basic Authentication header.

6.1. Identify Whether Redfish Service Supports Composition

Application code should always start at the root: `/redfish/v1/`

1. Read the Service Root Resource.
    1. Find the `CompositionService` property.
    2. Perform a GET on the URI given by that property.
    3. Look for the value of `ServiceEnabled` attribute to be true.

```
Client|                                                    | Redfish Service
   |---- GET /redfish/v1/CompositionService ----->|
   |<--- { ..., "ServiceEnabled": true, ... } <---|
```

6.2. Read the List of Resources Available for Composition

The client needs to understand the composition model reported by the Composition Service by reading the Resource Blocks and Resource Zones collections. This relationship will be used to execute the reported `UseCase` supported by the Redfish service described later in the Create a Composed Resource section.

1. Read the Resource Blocks.
    1. Perform a GET on the Composition Service URI.
    2. Look for the `ResourceBlocks` property.
    3. Perform a GET on that URI to get a list of all Resource Blocks.
    4. For accessing details about a particular Resource Block, perform a GET on the associated URI listed for a given entry in the `Members` array.

5. The `CompositionStatus` property in each Resource Block will identify the availablity of the Resource Block in composition requests.

    - Clients should take note of this when making decisions on what Resource Blocks to use in a composition request.
    - Depending on what is contained in the `CompositionStatus` property, a given Resource Block may not be currently available for composition.

```
{
 "@odata.type": "#ResourceBlockCollection.ResourceBlockCollection",
 "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks",
 "Name": "Resource Block Collection",
 "Members@odata.count": 9,
 "Members": [
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
ComputeBlock1" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
ComputeBlock2" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock3" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock4" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock5" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock6" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
DriveBlock7" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
NetworkBlock8" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
OffloadBlock9" }
 ]
}
```

2. Read the Resource Zones.

    1. Perform a GET on the Composition Service URI.
    2. Look for the `ResourceZones` property.
    3. Perform a GET on that URI to get a list of all Resource Zones.
    4. For accessing details about a particular Resource Zone, perform a GET on the associated URI listed for a given entry in the `Members` array.

```
{
 "@odata.type": "#ZoneCollection.ZoneCollection",
 "@odata.id": "/redfish/v1/CompositionService/ResourceZones",
 "Name": "Resource Zone Collection",
 "Members@odata.count": 2,
 "Members": [
     { "@odata.id": "/redfish/v1/CompositionService/ResourceZones/1" },
     { "@odata.id": "/redfish/v1/CompositionService/ResourceZones/2" }
 ]
}
```

3. Read the Capabilities for each Resource Zone.

   1. Perform a GET on each Resource Zone using the URI found in each entry of the `Members` array.

   2. Look for the `@Redfish.CollectionCapabilities` annotation in each Resource Zone.

      - The `UseCase` property will be used later when a client has determined what type of composition to create.
      - The `TargetCollection` property will be used later for making the composition request.

```
{
 "@odata.context": "/redfish/v1/$metadata#Zone.Zone",
 "@odata.type": "#Zone.v1_1_0.Zone",
 "@odata.id": "/redfish/v1/CompositionService/ResourceZones/1",
 "Id": "1",
 "Name": "Resource Zone 1",
 "Status": {},
 "Links": {},
 "@Redfish.CollectionCapabilities": {
     "@odata.type":
"#CollectionCapabilities.v1_0_0.CollectionCapabilities",
     "Capabilities": [
         {
             "CapabilitiesObject": { "@odata.id": "/redfish/v1/Systems/
Capabilities" },
             "UseCase":"ComputerSystemComposition",
             "Links": {
                 "TargetCollection": { "@odata.id": "/redfish/v1/
Systems" },
                 "RelatedItem": [
                     { "@odata.id": "/redfish/v1/CompositionService/
ResourceZones/1" }
                 ]
             }
         }
     ]
 }
}
```

4. Read each Capabilities Object.

    1. Perform a GET on the URI listed in the `CapabilitiesObject` property for each of the Capabilities.

```
{
 "@odata.context": "/redfish/
v1/$metadata#ComputerSystem.ComputerSystem",
 "@odata.type": "#ComputerSystem.v1_4_0.ComputerSystem",
 "@odata.id": "/redfish/v1/Systems/Capabilities",
 "Id": "Capabilities",
 "Name": "Capabilities for the Zone",
 "Name@Redfish.RequiredOnCreate": true,
 "Name@Redfish.SetOnlyOnCreate": true,
 "Description@Redfish.OptionalOnCreate": true,
 "Description@Redfish.SetOnlyOnCreate": true,
 "HostName@Redfish.OptionalOnCreate": true,
 "HostName@Redfish.UpdatableAfterCreate": true,
 "Boot@Redfish.OptionalOnCreate": true,
 "Boot": {
     "BootSourceOverrideEnabled@Redfish.OptionalOnCreate": true,
     "BootSourceOverrideEnabled@Redfish.UpdatableAfterCreate": true,
     "BootSourceOverrideTarget@Redfish.OptionalOnCreate": true,
     "BootSourceOverrideTarget@Redfish.UpdatableAfterCreate": true,
     "BootSourceOverrideTarget@Redfish.AllowableValues": [
         "None",
         "Pxe",
         "Usb",
         "Hdd"
     ]
 },
 "Links@Redfish.RequiredOnCreate": true,
 "Links": {
     "ResourceBlocks@Redfish.RequiredOnCreate": true,
     "ResourceBlocks@Redfish.UpdatableAfterCreate": true
 }
}
```

6.3. Create a Composed Resource

For building a composition request, the client can take the following steps:

1. List all Resource Blocks that belong to a particular Resource Zone by doing a GET on their collection URIs as described in the above example.
    - When reading the Resource Blocks, take note of the `CompositionStatus` property.
    - Depending on what is contained in the `CompositionStatus` property, a given Resource Block may not be currently available for composition.
2. (Optional) Reserve each Resource Block that has been identified for the composition request.
    - Perform a PATCH on each Resource Block with `Reserved` set to true.
    - This step should be done in scenarios where multiple clients may be making composition requests.
3. Identify the needs of a specific composition `UseCase`.
    1. Perform a GET on the desired Resource Zone.
    2. Find the matching `UseCase` value in the `@Redfish.CollectionCapabilities` annotation.
        - For example, look for the value `ComputerSystemComposition` if you are trying to compose a new Computer System from a specific list of Resource Blocks.
    3. Perform a GET on the URI found in the property `CapabilitiesObject`.
    4. Mark down all of the properties annotated with `RequiredOnCreate`.
        - These are the properties that need to be passed as part of the composition request.
    5. Mark down the `TargetCollection` URI.
        - This is the where the create (POST) request for the new composition is made.
4. Using all the properties that were annotated with `RequiredOnCreate`, build a create (POST) request body that will be sent to the `TargetCollection` URI.
    - In step 4 of the above example, only `Name` and `ResourceBlocks` found in `Links` are required.
    - The Redfish service may accept other properties as part of the request so they do not need to be updated later.
5. The `Location` HTTP header in the service response contains the URI of the composed resource.

General Flow Diagram:

```
Client  |                                                                   |
Redfish Service
        |---> GET /redfish/v1/CompositionService/ResourceZones/1 ------------>|
        |<--- { ..., "UseCase": "ComputerSystemComposition", ... } <----------|
        |                                                                   |
        |---> GET /redfish/v1/Systems/Capabilities -------------------------->|
        |     {   ...,                                                <----------|
        |         "Name@Redfish.RequiredOnCreate": true,                        |
        |         "ResourceBlocks@Redfish.RequiredOnCreate": true,              |
        |         ...                                                           |
        |<--- }                                                                 |
        |                                                                       |
        |           ( << Identify which Resource Blocks to use >> )             |
        |                                                                       |
        |-> GET /redfish/v1/CompositionService/ResourceBlocks/ComputeBlock2 ->|
        |<--- { ..., "CompositionState": "Unused", "Reserved": false ... } <--|
        |                                                                       |
        |-> PATCH /redfish/v1/CompositionService/ResourceBlocks/ComputeBlock2 |
        |   { "CompositionStatus": { "Reserved": true } } -------------------->|
```

Client Request Example:

```
POST /redfish/v1/Systems HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: <computed-length>
OData-Version: 4.0
{
    "Name": "Sample Composed System",
    "Links": {
        "ResourceBlocks": [
            { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
ComputeBlock0" },
            { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/DriveBlock2"
}
        ]
    }
}
```

Service Response Example:

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Content-Length: <computed-length>
Location: /redfish/v1/Systems/NewSystem
```

The above Client Request Example shows a composition request by the client being made to the Computer System Collection found at `/redfish/v1/Systems`. In the request, the client is creating a new Computer System using the Resource Blocks `ComputeBlock0` and `DriveBlock2`. In the above

Service Response Example, the service responded with a successful 201 response, and indicated that the new Computer System can be found at `/redfish/v1/Systems/NewSystem`.

6.4. Update a Composed Resource

If the Redfish service supports updating an existing composition, the client can update an already created composition through PUT/PATCH. This can be done by updating the `ResourceBlocks` array found in the composed resource. When using PATCH, the same array semantics apply as described in the Redfish Specification.

Client Request Example:

```
PATCH /redfish/v1/Systems/NewSystem HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: <computed-length>
OData-Version: 4.0
{
    "Links": {
        "ResourceBlocks": [
            {},
            {},
            { "@odata.id": "/redfish/v1/CompositionService/ResourceBlocks/
NetworkBlock8" }
        ]
    }
}
```

The above example will preserve the existing Resource Blocks in the composed resource for array elements 0 and 1, and it will add a the `NetworkBlock8` Resource Block to array element 2.

6.5. Delete a Composed Resource

The client can retire or decompose an already composed resource using DELETE.

Client Request Example:

```
DELETE /redfish/v1/Systems/NewSystem HTTP/1.1
```

The above example will request that the composed system called `NewSystem` be retired. When this happens, this will free the Resource Blocks being used by the system so that they can be used in future compositions. However, the `Reserved` flag found in the `CompositionStatus` for each Resource Block will remain in the same state; if a client is finished using the Resource Blocks, the client should set the

`Reserved` flag to false.

7. References

- "Composable System" and "Bladed Partitions" Mockups: http://redfish.dmtf.org/redfish/v1
- Composition Service Schema: http://redfish.dmtf.org/schemas/v1/CompositionService_v1.xml
- Resource Block Schema: http://redfish.dmtf.org/schemas/v1/ResourceBlock_v1.xml
- Resource Zone Schema: http://redfish.dmtf.org/schemas/v1/Zone_v1.xml
- Collection Capabilities Schema: http://redfish.dmtf.org/schemas/v1/CollectionCapabilities_v1.xml