What is HAL?

No, this is not a reference to the HAL 9000 from Clarke's *2001: A Space Odyssey*,

Quick Answer

HAL is an acronym for *Hypertext Application Language* and is designed to provide "...a set of conventions for expressing hyperlinks in either JSON or XML." HAL is an IANA registered media type. This enables RESTful APIs to return JSON* containing hypertext (aka hypermedia) which can be easily interpreted by machine and human agents. Because it is a registered media type, developers can specify **`application/hal+json`** as the value for the *Content-Type* response header.

** There is a HAL media type for both JSON and XML. However, we will limit our attention to JSON since that is what the SRP HTTP Framework uses by default.*

Digging Deeper

In our *What is REST?* article we introduced an important aspect of the *Uniform Interface* constraint known as HATEOAS. The intent of *HATEOAS* borrows the concept of HTML hyperlinks as a means of providing self-documented options to users. The difference is that HTML defines this as a standard which makes it possible for clients to interpret the hyperlinks automatically. Plain old JSON does not include any definition for hypermedia. This means any hypermedia included in a JSON object has no way of being self-interpreting. Clients cannot know the difference between a URI that is part of the static resource data (i.e., just plain media) versus a URI that clients can use to navigate to another resource (i.e., hypermedia).

HAL resolves this by extending JSON to include hypermedia and define the standard so clients can properly identify hypermedia within JSON objects. This is why the SRP HTTP Framework normally sets the *Content-Type* response header to **`application/hal+json`** instead of just `application/json`. It is important to note that HAL does not redefine JSON. This means clients can navigate and parse HAL content using the same rules as standard JSON.

Reserved Properties

HAL reserves two properties so that clients can interpret hypermedia in a resource object: ***_links*** and ***_embedded***. These properties are prefixed with underscores to avoid collision with properties that developers might use in actual static resource objects.

_links

This property contains one or more *sub-properties*. Each *sub-property* name defines its relation to the primary resource object. The *sub-property* values are themselves name/value pairs. Here is an example of a resource object containing a few link relations:

```
{
   "firstName":"James",
   "lastName":"Butt",
   "address":"6649 N Blue Gum St",
   "city":"New Orleans",
   "state":"LA",
   "zip":"70116",
   "county":"Orleans",
   "email":"jbutt@gmail.com",
   "_links":{
      "self":{
         "href":"https://www.examples.org/api/customers/1"
      },
      "collection":{
         "href":"https://www.examples.org/api/customers"
      },
      "openOrders":{
         "href":"https://www.examples.org/api/customers/1/orders?status=Open"
      }
   }
}
```

In the above resource object, there are three relations defined: *self*, *collection*, and *openOrders*. Each relation then includes an *href* sub-property with a URI value that allows the client to request that resource.

In the HAL specification draft, there SHOULD always be a *self* relation. This references the current resource object, as per the Web Linking RFC. Other relations can be defined by the developer as needed assuming they are meaningful to the resource. All relation names should be immutable. That is, once a relation name has been defined, clients should assume it will never change. The URI that a relation points to can change in order to maintain the decoupled and client/server independent intent of HATEOAS. Relations can be deprecated.

HAL intentionally defined link relations to be simple. To support the self-documenting and discoverability intent of a RESTful API, clients should be able to use the *OPTIONS* HTTP method to get a list of all supported HTTP methods for a given API. The list of all supported HTTP methods is expected to appear in the *Allow* response header. The SRP HTTP Framework handles this automatically.

_embedded

This property contains one or more *sub-properties*. Each *sub-property* name defines one or more embedded resource objects within the current resource object. These embedded resource objects (which are technically *sub-resources*) represent a partial or whole resource object that should have its own API (i.e., URI endpoint). The reason for including embedded resource objects will vary and are always based on the developer's needs. Embedded resource objects are typically added to provide one API with enough additional resource information to prevent the need for additional API calls. This is very common with *collection* APIs. For example, in the above resource object we identified a URI with a relation of *collection*. An inspection of the URI itself reveals that the endpoint is simply `/customers`. The absence of a unique identifier to a customer (e.g., `/customers/1`) is expected to mean "all customers", also referred to as "the *collection* of customers".

At a minimum, the `GET /customers` API should return all URIs related to each customer in the collection. However, perhaps a client wants to use this API as a way to provide visitors with a list of customers. If the client intends to display other customer properties (e.g., first and last name), then the client would need to make a `GET /customers/{id}` request for each URI. This could become a resource intensive process. It would be far better if the initial `GET /customers` request returned all the URIs along with partial resource objects which can be immediately consumed and used by the client. Here is an example of a resource object with one embedded resource:

```
{
   "_embedded":{
      "customers":[
         {
            "firstName":"Harrison",
            "lastName":"Haufler",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/266"
               }
            }
         },
         {
            "firstName":"Haydee",
            "lastName":"Denooyer",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/271"
               }
            }
         },
         {
            "firstName":"Heike",
            "lastName":"Berganza",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/254"
               }
            }
         },
         {
            "firstName":"Helga",
            "lastName":"Fredicks",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/202"
               }
            }
         },
         {
            "firstName":"Herman",
            "lastName":"Demesa",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/125"
               }
            }
```

```
         },
         {
            "firstName":"Herminia",
            "lastName":"Nicolozakes",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/287"
               }
            }
         },
         {
            "firstName":"Hillary",
            "lastName":"Skulski",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/186"
               }
            }
         },
         {
            "firstName":"Howard",
            "lastName":"Paulas",
            "_links":{
               "self":{
                  "href":"https://www.examples.org/api/customers/132"
               }
            }
         }
      ]
   }
}
```

Here are a few notes regarding the above resource object and embedded resources in general:

- The name of the embedded resource in our example is *customers*.
- Other embedded resource names can be included. Each embedded resource name is a *sub-property* of the *_embedded* property.
- Embedded resources can be singular (i.e., the value is just a resource object) or plural (i.e., the value is an array of resource objects). If the embedded resource object is singular by definition, then just set the value to a resource object. Otherwise, use an array even if there is only one embedded resource object. Otherwise, adding an array later on could break clients.
- Each embedded resource object should also include its own *self* relation link (which is identified by the *_links* property).

In Summary

While HAL is not the only IANA registered media type that provides standards for including hypermedia in JSON objects, it is one of the most popular and easiest to implement. For this reason, the SRP HTTP Framework provides the *AddLinkRelation* and *AddEmbeddedResources* services to make it much easier to create *HATEOAS* APIs.