HATEOAS and Hypermedia in ASP.NET Core Web APIs

Hypermedia as the Engine of Application State (HATEOAS) is a key REST constraint that makes APIs **self-descriptive and discoverable**. Rather than relying on out-of-band documentation, a hypermedia-driven API includes links and actions in its responses so clients know what they can do next learn.microsoft.comlearn.microsoft.com. For example, if a purchase order is canceled, the server simply omits or disables the “submit” link for that resource, so clients automatically know the action isn’t allowedlearn.microsoft.com. In practice, the server returns **data + hypermedia artifacts** (links/forms) based on the current application statelearn.microsoft.com. This decouples clients from server details and makes APIs more evolvable.

By default ASP.NET Core returns JSON (media type `application/json`) and does not include link semantics. To add hypermedia, we choose a **media type** that embeds links. The most common JSON-based formats are **HAL**, **JSON-LD**, and **Siren**. Each defines conventions (properties like `_links` or `@context`) to include URLs and actions. APIs advertise these formats via the `Accept` header (e.g. `application/hal+json`, `application/ld+json`, `application/vnd.siren+json`) so clients know which hypermedia style to uselearn.microsoft.com. Below we explore each format and show how to implement them in C# ASP.NET Core APIs.

Hypermedia Formats: HAL, JSON-LD, Siren

- **HAL (Hypertext Application Language)** – A simple JSON convention for embedding links. Responses include an `_links` object where each link has a **rel** and **href**. Resources may also embed other resources under `_embedded`. HAL is lightweight and widely supportedlearn.microsoft.comgithub.com. For example:

```
{ "_links": {      "self": { "href": "/orders" },
            "next": { "href": "/orders?page=2" } },
  "count": 2,
  "_embedded": {"orders": [
            {"id": 4,
            "item": "Foo",
            "_links": { "self": { "href": "/orders/4" } }
            },
            {
            "id": 5,
            "item": "Bar",
            "_links": { "self": { "href": "/orders/5" } }
            }
            ]}}
```

In the example above, `_links.self` points to the current resource, and `_links.next` points to the next page. Each embedded order also has its own `_links.self`. HAL's conventions standardize these link objectsgithub.com. When using HAL, the media type is `application/hal+json`learn.microsoft.com.

JSON-LD (JSON for Linked Data) – A JSON format that adds **Linked Data** semantics using `@`-prefixed keywords. JSON-LD lets you embed a context (`@context`) that defines the meaning of properties, and use `@id` and `@type` for resource identifiers and types. It's often used with Schema.org or other vocabularies. For example:

```
{
  "@context": "https://schema.org/",
  "@id": "https://api.example.com/people/1",
  "@type": "Person",
  "name": "Alice",
  "knows": "https://api.example.com/people/2"
}
```

Here `@context` maps terms to JSON-LD definitions, `@id` is the resource URI, and `knows` points to another person's URI. JSON-LD is **lightweight and human-readable**, and is ideal for REST servicesjson-ld.org. Clients request it with `Accept: application/ld+json` and can follow `@id` links.

Siren – A hypermedia specification focused on **entities, actions, and links**. A Siren entity JSON has fields like `"class"`, `"properties"`, `"entities"` (sub-entities), `"actions"`, and `"links"`. For example:

```
{"class": ["order"],
  "properties": { "orderNumber": 42, "status": "pending" },
  "entities": [ {
        "class": ["items", "collection"],
        "rel": ["http://x.io/rels/order-items"],
        "href": "http://api.example.com/orders/42/items"
        },       {         "class": ["info", "customer"],
        "rel": ["http://x.io/rels/customer"],
        "properties": { "customerId": "pj123", "name": "Peter Joseph" },
        "links": [
        { "rel": ["self"], "href": "http://api.example.com/customers/pj123" } ]         } ],
```

```
"actions": [{
        "name": "add-item",
        "title": "Add Item",
        "method": "POST",
        "href": "http://api.example.com/orders/42/items",
        "type": "application/x-www-form-urlencoded",
        "fields": [
        { "name": "orderNumber", "type": "hidden", "value": "42" },
        { "name": "productCode", "type": "text" },
        { "name": "quantity", "type": "number" }
        ]
        } ],
"links": [
        { "rel": ["self"], "href": "http://api.example.com/orders/42" },
        { "rel": ["next"], "href": "http://api.example.com/orders/43" }]}
```

- In Siren, `"class"` denotes the entity type, `"properties"` holds its state, `"entities"` may embed or link related entities, and `"actions"` enumerate possible state-changing operations. Siren supports both links **and** forms (actions), making it powerful for full hypermedia-driven APIsgithub.com. Clients use `Accept: application/vnd.siren+json` to receive Siren.

Each of these media types has conventions for links. Using them makes the API **self-discoverable**, since clients can parse responses to find URLs for next actions. In ASP.NET Core you can implement any of these: either by **manually shaping the JSON**, or by using output formatters/libraries. In the sections below we show examples of implementing HAL, JSON-LD, and Siren in a C# Web API, including dynamic link generation and content negotiation.

Choosing the Right Response Format

A simple JSON response is ideal when your API is used internally or when clients already know all available endpoints. It keeps payloads small and performance optimal.

Use **HAL** when you want clients to discover related resources and navigate your API effortlessly. HAL adds a `_links` section to each response, enabling pagination (via `next` and `prev` links) and resource relations without extra overhead.

Choose **JSON-LD** for public data exposure, especially if you need semantic context or integration with Linked Data platforms. JSON-LD embeds a `@context` that defines terms, making your API data machine-readable by search engines and semantic consumers.

Opt for **Siren** when your API surfaces complex domain operations. Siren not only provides links, but also describes available actions (forms) on each resource. This makes it perfect for clients that need to discover and execute state-changing operations dynamically.

By selecting the appropriate format:

- **Plain JSON** keeps things lightweight when hypermedia isn’t necessary.
- **HAL** provides basic discoverability and paging support.
- **JSON-LD** enables semantic web and search engine friendliness.
- **Siren** supports rich, action-driven workflows within your API.

Choose based on your application’s needs for discoverability, semantic richness, or supported client workflows.

Implementing HAL in ASP.NET Core

To return HAL, include an `_links` object in your JSON. You can do this manually or use a helper library. For example, create a model with a `_links` property (in C#, we use `JsonProperty` to emit `_links`):

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    [JsonProperty("_links")]
    public Dictionary<string, object> Links { get; set; } = new Dictionary<string, object>();
}
```

In a controller, use **named routes** and `Url.Link` to build URLs dynamically. For instance:

```
[ApiController]

[Route("api/[controller]")]

public class ProductsController : ControllerBase

{       // GET api/products/5

        [HttpGet("{id}", Name = "GetProduct")]

        [Produces("application/hal+json")]

        public IActionResult GetProduct(int id)

        {

        // Example data fetch (replace with real data access)

        var product = new Product { Id = id, Name = "Widget" };
```

```
// Dynamically generate hypermedia links

product.Links["self"] = new { href = Url.Link("GetProduct", new { id = product.Id }) };

product.Links["update"] = new { href = Url.Link("UpdateProduct", new { id = product.Id }) };

product.Links["delete"] = new { href = Url.Link("DeleteProduct", new { id = product.Id }) };


return Ok(product);

}
```

```
// PUT api/products/5
[HttpPut("{id}", Name = "UpdateProduct")]
public IActionResult UpdateProduct(int id, [FromBody] Product input)
{
// Update logic here...
return NoContent();
}
// DELETE api/products/5
[HttpDelete("{id}", Name = "DeleteProduct")]
public IActionResult DeleteProduct(int id)
{
// Delete logic here...
return NoContent();    }}
```

In the above, `Url.Link("GetProduct", new { id = product.Id })` produces an absolute URL for the named route `GetProduct`. ASP.NET Core's routing and URL helpers let you change routes without hardcoding URLs (similar to `UrlHelper` in older Web API)learn.microsoft.com. The result returned to the client will look like:

```
{
  "Id": 5,
  "Name": "Widget",
  "_links": {
      "self": { "href": "https://api.example.com/api/products/5" },
      "update": { "href": "https://api.example.com/api/products/5" },
      "delete": { "href": "https://api.example.com/api/products/5" }
  }
}
```

Note the `_links` with `self`, `update`, `delete`. This follows the HAL conventions (using `_links` for URLs)learn.microsoft.com.

To fully enable HAL with content negotiation, you can **add a HAL formatter/library**. For example, the daxnet/hal library for .NET Core provides middleware that automatically wraps responses in HAL. After installing the NuGet, enable it in `Program.cs`:

```
builder.Services.AddHalSupport();
```

This adds an output formatter for `application/hal+json` (by default it assumes your ID property is `Id`, but you can configure it)github.com. Then your controllers can return plain objects or lists, and the HAL support will automatically inject `_links` and `_embedded` fields. For example, if you return a list of `MeetingRoom` objects, HAL support will output HAL-formatted JSON with links as shown heregithub.com:

```
{
  "_links": { "self": { "href": "http://localhost/api/MeetingRooms/get-by-name/RoomA" } },
  "count": 1,
  "_embedded": {
          "meetingRooms": [
          {
          "id": 4,
          "name": "RoomA",
          "seats": 14,
          "_links": { "self": { "href": "http://localhost/api/MeetingRooms/4" } }
          }
          ]
  }
}
```

By using either manual shaping or a HAL library, clients requesting `application/hal+json` will receive HAL-style JSON. The `AddHalSupport()` method handles the boilerplate so you don't have to set `_links` everywheregithub.com. (If not using a library, ensure you set the `Content-Type` or use `[Produces("application/hal+json")]` so clients know it's HAL.)

Summary (HAL): Include an `_links` object with named relations. Use `Url.Link` or `LinkGenerator` to build dynamic URLs. You can implement it yourself or use libraries like daxnet/hal or older **WebApi.Hal** (for .NET Core) to format your objects to HALgithub.comgithub.com.

Implementing JSON-LD in ASP.NET Core

JSON-LD is simply JSON with special `@` fields for Linked Data. You can hand-craft JSON-LD or use a library like json-ld.net. The key parts are:

- `@context`: Defines the vocabulary/terms (often a URL or object).
- `@id`: The URI of the current resource.
- `@type`: (Optional) the type of the resource.
- Other fields can be plain JSON, but you can also use them to link to other resources.

For example, consider a `Person` resource. You might have:

```
public class Person
{
        [JsonProperty("@context")]
        public string Context { get; set; } = "https://schema.org/";
        [JsonProperty("@id")]
        public string Id { get; set; }
        [JsonProperty("@type")]
        public string Type { get; set; } = "Person";
        public string name { get; set; }
        public int age { get; set; }
        public string knows { get; set; }
}
```

And in the controller:

```
[HttpGet("{id}", Name = "GetPerson")]
[Produces("application/ld+json")]
public IActionResult GetPerson(int id)
{       // Example data (replace with real data access)
        var person = new Person {
        Id = Url.Link("GetPerson", new { id }),  // the @id of this resource
        name = "Alice",
        age = 30,
        knows = Url.Link("GetPerson", new { id = 2 })  // link to another person
        };
        return Ok(person);
}
```

This will produce JSON like:


```
{
  "@context": "https://schema.org/",
  "@id": "https://api.example.com/api/people/1",
  "@type": "Person",
  "name": "Alice",
  "age": 30,
  "knows": "https://api.example.com/api/people/2"
}
```

Here `@context` points to Schema.org's definitions (for example), and `@id` is the canonical URL for Alice. The `"knows"` field is just another property; in JSON-LD it's interpreted as linking to another `Person`. By including these fields, the API is using JSON-LD hypermediajson-ld.org. Clients can fetch this with `Accept: application/ld+json` to get JSON-LD, or use `application/json` if you choose to also emit it as plain JSON.

(You can also use JSON-LD framing or context expansion, but the basic approach is embedding URIs with `@id` or plain property values that are URIs. The ASP.NET serializer will emit these fields as shown. Optionally, you might include `@context` as an object if you want to define custom terms inline.)

Summary (JSON-LD): Create JSON objects with `@context`, `@id`, etc., and return them with content type `application/ld+json`. This ensures clients see it as JSON-LD. Because JSON-LD is standard JSON plus metadata, you don't need a special formatter—just shape your objects appropriately. The JSON-LD website confirms it's a **"lightweight Linked Data format"** based on JSONjson-ld.org.

Implementing Siren in ASP.NET Core

For Siren, you build a JSON structure with keys `"class"`, `"properties"`, `"entities"`, `"actions"`, and `"links"`. There's no built-in ASP.NET support for Siren, but you can hand-craft the JSON or use a helper library like **FluentSiren** or **SirenDotNet**. Here's a simple manual example:

```
[HttpGet("{id}", Name = "GetOrder")]
[Produces("application/vnd.siren+json")]
public IActionResult GetOrder(int id)
{
    // Example data
    var order = new {
    @class = new[] { "order" },
    properties = new { orderNumber = 42, status = "pending" },
```

```
entities = new object[] {
new {
        @class = new[] { "items", "collection" },
        rel = new[] { "http://x.io/rels/order-items" },
        href = Url.Link("GetOrderItems", new { orderId = id })
}
},
actions = new object[] {
new {
        name = "add-item",
        title = "Add Item",
        method = "POST",
```

```
        href = Url.Link("AddOrderItem", new { orderId = id }),
        type = "application/x-www-form-urlencoded",
        fields = new[] {
                new { name = "orderNumber", type = "hidden", value = id.ToString() },
                new { name = "productCode", type = "text" },
                new { name = "quantity", type = "number" }
        }
        }
        },
        links = new[] {
        new { rel = new[] { "self" }, href = Url.Link("GetOrder", new { id }) },
        new { rel = new[] { "next" }, href = Url.Link("GetOrder", new { id = id + 1 }) }
        }
        };
        return Ok(order);
}
```

This produces a Siren JSON like:

```
{
  "class": ["order"],
  "properties": { "orderNumber": 42, "status": "pending" },
  "entities": [
        {
        "class": ["items", "collection"],
        "rel": ["http://x.io/rels/order-items"],
        "href": "http://api.example.com/orders/42/items"
        }
  ],
```

```
"actions": [
        {
        "name": "add-item",
        "title": "Add Item",
        "method": "POST",
        "href": "http://api.example.com/orders/42/items",
        "type": "application/x-www-form-urlencoded",
        "fields": [
        { "name": "orderNumber", "type": "hidden", "value": "42" },
        { "name": "productCode", "type": "text" },
        { "name": "quantity", "type": "number" }
        ]
        }
],
"links": [
        { "rel": ["self"], "href": "http://api.example.com/orders/42" },
        { "rel": ["next"], "href": "http://api.example.com/orders/43" }]}
```

Key points:

- `"class"` is the entity type (like `order`).
- `"properties"` holds fields.
- `"entities"` can embed or link sub-resources.
- `"actions"` define possible transitions (forms).
- `"links"` is an array of link objects (`rel` and `href`).

The above code uses `Url.Link(...)` to generate dynamic URIs. You must set `Content-Type: application/vnd.siren+json` (e.g. via `[Produces]` or by checking `Accept`). Siren has libraries (e.g. FluentSiren, SirenDotNet) to help build these structures in code. Derek Comartin's blog notes **FluentSiren** and **SirenDotNet** among a few .NET options for Sirencodeopinion.com. Alternatively, you can simply serialize an anonymous object as shown.

Summary (Siren): Build a JSON object with keys `class`, `properties`, `entities`, `actions`, and `links` according to the Siren specgithub.com. Use `Url.Link` for URLs. There's no default ASP.NET Core formatter, so clients must specify `Accept: application/vnd.siren+json`, or you can manually set `Content-Type`. Libraries like **FluentSiren** or **SirenDotNet** can simplify creation, but simple objects work too.

Dynamic Link Generation & Content Negotiation

A crucial part of HATEOAS is **dynamic links**: you generate URLs at runtime instead of hardcoding. In ASP.NET Core, use `Url.Link` or `Url.Action` with named routes. In the earlier examples, `Url.Link("GetProduct", new { id = product.Id })` generated the URL for the product resource. This prevents broken links if routes change, and ensures consistencylearn.microsoft.com. You should give your routes names (e.g. `[HttpGet("{id}", Name="GetProduct")]`) so they can be referenced.

Content negotiation is also important. ASP.NET Core will, by default, respond in JSON (`application/json`). To support hypermedia formats, ensure your client's `Accept` header includes the desired media type (e.g. `application/hal+json`, `application/ld+json`, or `application/vnd.siren+json`). You can annotate actions with `[Produces(...)]` to indicate supported formats, or configure output formatters. For instance, returning `Ok(object)` will use `ObjectResult` and check `Accept`; if `application/hal+json` is requested and you have HAL support enabled, it will format accordinglylearn.microsoft.comlearn.microsoft.com. Without a custom formatter, you can still return the objects manually shaped as above and set `Content-Type` to the custom media type.

In summary, to make your ASP.NET Core Web API hypermedia-rich:

- **Define resource models** that include link structures (e.g. `_links` for HAL, or Siren fields, or JSON-LD context).
- **Generate URLs dynamically** using `Url.Link`/`Url.Action` or `LinkGenerator` so you always serve correct URIslearn.microsoft.com.
- **Use the correct media type** in `Accept`/`Content-Type`. Clients should request e.g. `application/hal+json` to get HAL or `application/ld+json` for JSON-LD.
- Optionally, **use libraries** or custom output formatters. For HAL, libraries like **daxnet/hal (HAL support)** or **WebApi.Hal** can automate link injectiongithub.com. For Siren, check out **FluentSiren** or **SirenDotNet**codeopinion.com. For JSON-LD, you can use [json-ld.net] or simply shape your JSON as shown.
- **Document link relations** (the meaning of each `rel`) according to your API's design. Standard relations like `self` or `next` have common meanings, and you can use URIs or CURIEs for others (as HAL does with curies).

By following these patterns, your API responses will guide clients through the available actions automatically, achieving true RESTful, discoverable APIs. The API **literally includes its own "map" of operations**, in the form of hypermedia linkslearn.microsoft.comgithub.com.

References: Official Microsoft docs and industry resources discuss hypermedia in ASP.NET Core (see [Microsoft HAL docs][9] and this MSDN articlelearn.microsoft.com). The [HAL spec][9], [JSON-LD site][11], and [Siren spec][12] define the formats. Libraries like daxnet's HAL support and FluentSiren can simplify implementationgithub.comcodeopinion.com. The ASP.NET Core [formatting docs][34] explain content negotiation, and blogs/QL articles show examples of adding HATEOAS in C# (e.g. Code Mazegithub.com).