CHAPTER 10

The Hypermedia Zoo

There are a lot of hypermedia document formats in active use. Some are designed for very specialized purposes—the people who use them may not even think of them as hypermedia formats. Other hypermedia formats are in such common usage that people don't really think about them at all. In this chapter, I'll take you on an educational tour of a "zoo" containing the most popular and most interesting hypermedia formats.

I won't be going into a lot of technical detail. Any one of these formats probably isn't the one you want to use, and I've covered many of them earlier in the book. Many of the formats are still under active development, and their details might change. If you're interested in one of the zoo's specimens, the next step is to read its formal specification.

My goal is to give you a sense of the many forms hypermedia can take, and to show how many times we've tackled the basic problems of representing it. The hypermedia zoo is so full that you probably don't need to define a brand new media type for your API. You should be able to pick an existing media type and write a profile for it.

I've organized the hypermedia zoo along the lines of my introduction to hypermedia. There's a section for domain-specific formats (a la Chapter 5), a section for formats whose primary purpose is to implement the collection pattern (a la Chapter 6), and a section for general hypermedia formats (a la Chapter 7).

For formats like Collection+JSON, which I've already covered in some depth, I'll briefly summarize the format and point you to the earlier discussion. There are a few hypermedia formats that I won't discuss in this chapter, because they take different approaches to REST than the one I've advocated so far in this book. I'll cover RDF and its descendants in Chapter 12, and CoRE Link Format in Chapter 13.

Domain-Specific Formats

These media types are designed to represent problems in one particular domain. Each defines some very specific application semantics, and although you might be able to use them to convey different semantics, it's probably a bad idea.

Maze+XML

- **Media type:** `application/vnd.amundsen.maze+xml`
- **Defined in:** personal standard
- **Medium:** XML
- **Protocol semantics:** navigation using GET links
- **Application semantics:** maze games
- **Covered in:** Chapter 5

Maze+XML defines XML tags and link relations relating to mazes, cells in mazes, and the connections between cells. Figure 10-1 gives the state diagram of its protocol semantics.

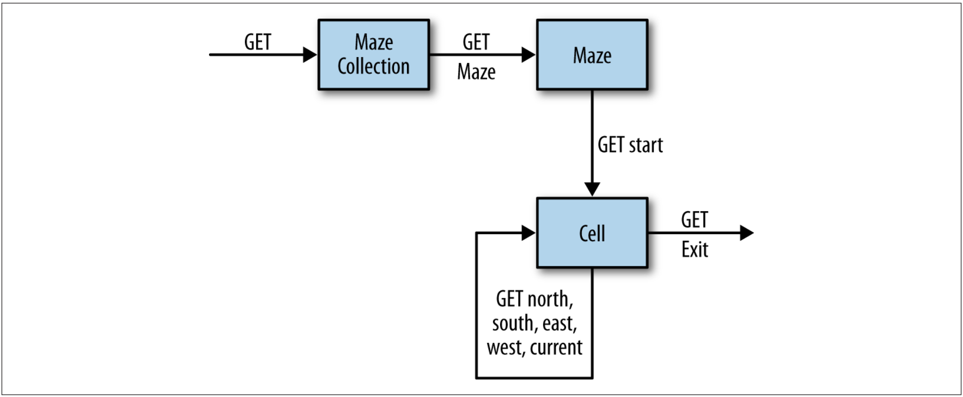


Figure 10-1. The protocol semantics of Maze+XML

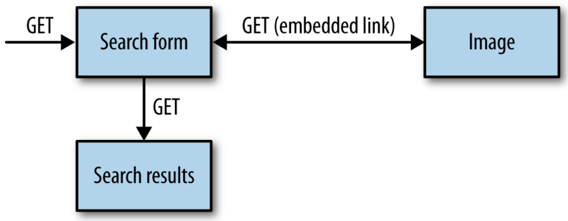
Maze+XML defines a `<link>` tag that takes a link relation and defines a safe state transition; that is, it allows the client to make a GET request. You can extend Maze+XML by bringing in custom link relations, or by defining extra XML tags. Since it's an XML format, you could also use XForms (q.v.) to represent unsafe state transitions.

I don't seriously recommend using Maze+XML, even if you happen to be making a maze game. It's just an example, and I'm putting it first to serve as an example of how I judge hypermedia formats.

OpenSearch

- **Media type:** `application/opensearchdescription+xml` (pending registration)
- **Defined in:** consortium standard
- **Medium:** XML
- **Protocol semantics:** searching using GET
- **Application semantics:** search queries
- **Covered in:** Chapter 6

OpenSearch is a standard for representing search forms. It can be used standalone, or incorporated into another API using the `search` link relation. Its state diagram looks like this:



Here's a simple OpenSearch representation. The destination of an OpenSearch form (the `template` attribute of its `<Url>` tag) is a string similar to a URI Template (RFC 6570), though it doesn't have all of URI Template's features:

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
 <ShortName>Name search</ShortName>
 <Description>Search the database by name</Description>
 <Url type="application/atom+xml" rel="results"
   template="http://example.com/search?q={searchTerms}"/>
</OpenSearchDescription>
```

OpenSearch does not define a way to represent the *results* of a search. You should use whatever list format fits in with your main representation format.

Problem Detail Documents

- **Media type:** `application/api-problem+json`
- **Described in:** Internet-Draft "draft-nottingham-http-problem"
- **Medium:** JSON (with rules for automatically converting to XML)
- **Protocol semantics:** navigation with GET

- **Application semantics:** error reports

A problem detail document describes an error condition. It uses structured, human-readable text to add custom semantics to HTTP's status codes. It's a simple JSON format designed to replace whatever one-off format you were thinking of designing to convey your error messages.

Like most JSON-based hypermedia documents, a problem detail takes the form of a JSON object. Here's a document that might be served along with an HTTP status code of 503 (`Service Unavailable`):

```
{
 "describedBy": "http://example.com/scheduled-maintenance",
 "supportId": "http://example.com/maintenance/outages/20130533",
 "httpStatus" : 503
 "title": "The API is down for scheduled maintenance.",
 "detail": "This outage will last from 02:00 until 04:30 UTC."
}
```

Two of these properties are defined as hypermedia links. The `describedBy` property is a link to a human-readable explanation of the representation.[1]

The `supportId` property is a URL representing *this particular instance* of the problem. There's no expectation that the end user will find anything at the other end of this URL. It might be an internal URL for use by the API support staff, or it might be a URI, a unique ID that doesn't point to anything in particular.

The `describedBy` and `title` properties are required; the rest are optional. You can also add extra properties specific to your API.

SVG

- **Media type:** `image/svg+xml`
- **Medium:** XML
- **Protocol semantics:** the same as XLink
- **Application semantics:** vector graphics

SVG is an image format. Unlike a JPEG, which represents an image on the pixel level, an SVG image is made up of shapes. SVG includes a hypermedia control that lets different parts of an image link to different resources.

1. `describedBy` is an IANA-registered link relation that's a more general version of `profile`. A resource is `describedBy` any resource that sheds *any* light on its interpretation.

That hypermedia control is an `<a>` tag that has the same function as HTML's `<a>` tag. Here's a simple SVG representation of a cell in Chapter 5's maze:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">

  <rect x="100" y="80" width="100" height="50" stroke="black" fill="white"/>
  <text x="105" y="105" font-size="10">Foyer of Horrors</text>

  <a xlink:href="/cells/I" xlink:arcrole="http://alps.io/example/maze#north">
   <line x1="150" y1="80" x2="150" y2="40" stroke="black"/>
   <text x="130" y="38" font-size="10">Go North!</text>
  </a>

  <a xlink:href="/cells/O" xlink:arcrole="http://alps.io/example/maze#east">
   <line x1="200" y1="105" x2="240" y2="105" stroke="black"/>
   <text x="240" y="107" font-size="10">Go East!</text>
  </a>

  <a xlink:href="/cells/M" xlink:arcrole="http://alps.io/example/maze#west">
   <line x1="100" y1="105" x2="60" y2="105" stroke="black"/>
   <text x="18" y="107" font-size="10">Go West!</text>
  </a>

</svg>
```

Figure 10-2 shows how a client might render this document.

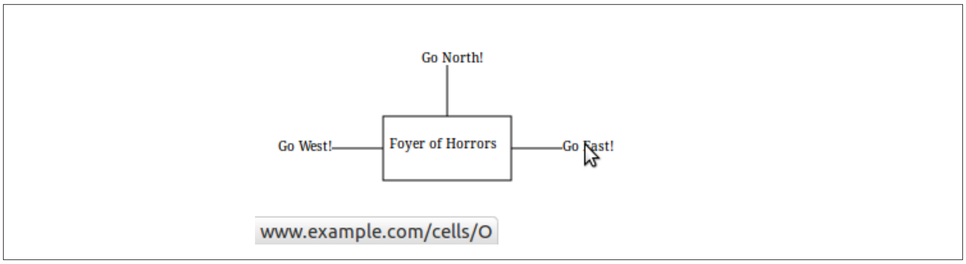


Figure 10-2. The SVG representation of a maze cell

SVG makes a good alternative to HTML for building mobile applications. SVG can also be combined with HTML 5: just stick an `<svg>` tag into HTML markup to get an inline SVG image.

SVG's `<a>` tag doesn't actually define any hypermedia capabilities. It's just a placeholder tag for XLink's `role` and `href` attributes (q.v.). Since SVG is an XML format, you can also add XForms forms (q.v.) to SVG, and get protocol semantics comparable to HTML's. This is not as useful as embedding SVG into HTML, since it requires a client that understands both SVG and XForms.

VoiceXML

- **Media type:** `application/voicexml+xml`
- **Defined in:** W3C open standard, with extensions
- **Medium:** XML
- **Protocol semantics:** GET for navigation; arbitrary state transitions through forms: GET for safe transitions, POST for unsafe transitions
- **Application semantics:** spoken conversation

In Chapter 5, I made an analogy between an HTTP client navigating a hypermedia API and a human being navigating a phone tree. Well, a lot of those phone trees are actually implemented on the backend as hypermedia APIs. The representation format they use is VoiceXML.

Here's one possible VoiceXML representation of a cell in Chapter 5's maze game:

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.1">
 <menu>
  <prompt>
   You are in the Foyer of Horrors. Exits are: <enumerate/>
  </prompt>

  <choice next="/cells/I">
   North
  </choice>

  <choice next="/cells/M">
   East
  </choice>

  <choice next="/cells/O">
   West
  </choice>

  <noinput>Please say one of <enumerate/></noinput>
  <nomatch>You can't go that way. Exits are: <enumerate/></nomatch>
 </menu>
</vxml>
```

If you're playing the maze game over the phone, you'll never see this representation directly. The VoiceXML "browser" lives on the other end of the phone line. When it receives this representation, it handles the document by reading the `<prompt>` aloud to you: "You are in the Foyer of Horrors. Exits are: north, east, west."

Each `<choice>` tag is a hypermedia link. The browser waits for you to activate a link by saying something. It uses speech recognition to figure out which link you're activating. There's a validation step: if you say nothing, or you say something that doesn't map onto one of the links, the browser reads you an error message (either `<noinput>` or `<nomatch>`) and waits for input again.

Once you manage to activate a link, the browser makes a GET request to the URL mentioned in the corresponding `next` attribute. The server responds with a new VoiceXML representation, and the browser processes the representation and tells you which maze cell you're in now.

The `<menu>` tag is only the simplest of VoiceXML's hypermedia controls. There's also a `<form>` tag that uses a speech recognition grammar to drive a GET or POST request based on what you tell it. Here's a VoiceXML form for flipping the mysterious switches I defined in Chapter 7:

```
<form id="switches">
 <grammar src="command.grxml" type="application/srgs+xml"/>

 <initial name="start">
  <prompt>
   There is a red switch and a blue switch here. The red switch is
   up and the blue switch is down.

   What would you like to do?
  </prompt>
 </initial>

 <field name="command">
  <prompt>
   Would you like to flip the red switch, flip the blue switch, or
   forget about it?
  </prompt>
 </field>

 <field name="switch">
  <prompt>
   Say the name of a switch.
  </prompt>
 </field>

 <filled>
  <submit next="/cells/I" method="POST" namelist="command switch"/>
 </filled>
</form>
```

The `<grammar>` tag is an inline link analogous to an HTML `<img>` or `<script>` tag. It automatically imports a document written in a format set down by the W3C's Speech

Recognition Grammar Specification.[2] I won't show the SRGS file here, because SRGS is not a hypermedia format. Suffice to say that when you say the words "flip the red switch," or "forget about it," the SRGS grammar is what allows the VoiceXML browser to transform those words into a set of key-value pairs that match the form fields `command` and `switch`:

```
command=flip
switch=red switch
```

Once the fields are filled in with values obtained through speech recognition, the `<submit>` tag tells the VoiceXML browser how to format an HTTP POST request. It looks just like an HTML form submission:

```
POST /cells/I HTTP/1.1
Content-Type: application/x-www-form-urlencoded

command=flip&switch=red%20switch
```

A VoiceXML document resembles nothing so much as programming language code. VoiceXML uses idioms from programming to represent the flow of conversation through a dialog tree: `<goto>` to jump from one part of the dialog to another, `<if>` to represent a conditional, and even `<var>` to assign a value to a variable.

Collection Pattern Formats

The three standards in this section have similar application and protocol semantics, because they all implement the collection pattern I laid out in Chapter 6. In the collection pattern, certain resources are designated "item" resources. An item usually responds to GET, PUT, and DELETE, and its representation focuses on representing structured bits of data. Other resources are designated "collection" resources. A collection usually responds to GET and POST-to-append, and its representation focuses on linking to item resources.

These three standards take different approaches to the collection pattern; they may not use the terms "collection" or "item," but they all do pretty much the same thing.

Collection+JSON

- **Media type:** `application/vnd.amundsen.collection+json`
- **Defined in:** personal standard
- **Medium:** JSON

2. Defined here.

- **Protocol semantics:** collection pattern (GET/POST/PUT/DELETE), plus searching (using GET)
- **Application semantics:** collection pattern ("collection" and "item")
- **Covered in:** Chapter 6

Collection+JSON was designed as a simple JSON-based alternative to the Atom Publishing Protocol (q.v.). It's a formalized, hypermedia-aware version of the API developers tend to design their first time through the process. Figure 10-3 shows its protocol semantics.

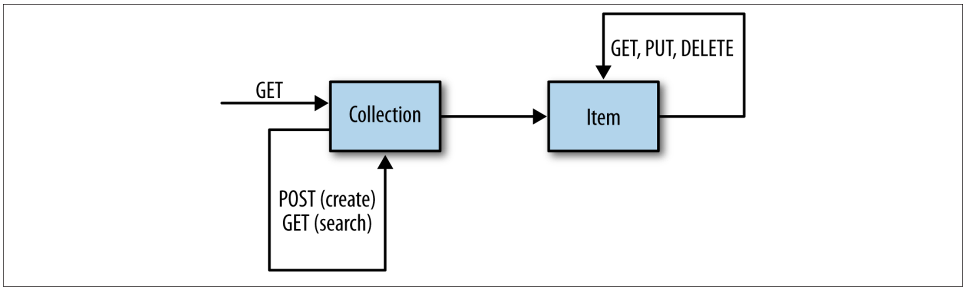


Figure 10-3. The protocol semantics of Collection+JSON

The Atom Publishing Protocol

- **Media types:** `application/atom+xml`, `application/atomsvc+xml`, and `application/atomcat+xml`
- **Defined in:** RFC 5023 and RFC 4287
- **Medium:** XML
- **Protocol semantics:** collection pattern (GET/POST/PUT/DELETE); well-defined extensions add searching and other forms of navigation, all using GET links or forms
- **Application semantics:** collection pattern (`feed` and `entry`); entries have the semantics of blog posts (`author`, `title`, `category`, etc.); an entry that is not an Atom document (e.g., a binary graphic) is split into a binary `Media Entry` and an Atom `Entry` that contains metadata
- **Covered in:** Chapter 6

The original API standard, AtomPub pioneered the collection pattern and the RESTful approach to APIs in general. As an XML-based standard in a field now dominated by JSON representations, AtomPub now looks somewhat old-fashioned, but it inspired several other standards and link relations that can be used with other hypermedia formats. Figure 10-4 shows its protocol semantics.

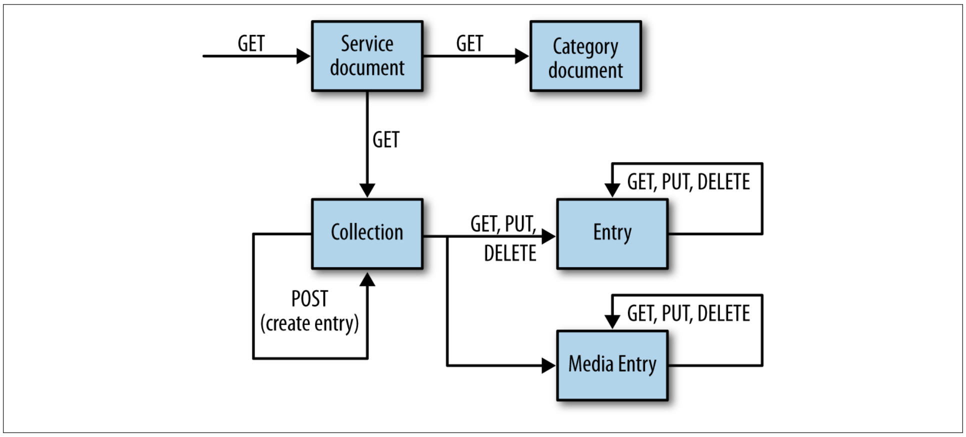


Figure 10-4. The protocol semantics of AtomPub

Although Atom's application semantics imply that it should be used only for news-feed applications like blogging and content management APIs, the standard is very extensible. Perhaps the most notable extension is the Google Data Protocol, the foundation of Google's API platform. Google adds domain-specific tags to AtomPub to describe the application semantics of each of its sites. An Atom feed becomes a collection of videos (the YouTube API) or a collection of spreadsheet cells (Google Spreadsheets).

If you think your application semantics won't fit into the collection pattern, a look at Google's API directory may convince you otherwise. The Google Data Protocol also defines a JSON equivalent to AtomPub's XML representations, though this is a fiat standard, not something you're invited to reuse.

Several open standards define AtomPub extensions, including the Atom Threading Extensions and the `deleted-entry` element. I covered these in Chapter 6.

OData

- **Media type:** `application/json;odata=fullmetadata`
- **Defined in:** open standard in progress
- **Medium:** JSON for some parts, XML for others

- **Protocol semantics:** modified collection pattern (GET/POST/PUT/DELETE) with PATCH for partial updates and GET for queries; arbitrary state transitions with forms (GET for safe transitions, and POST for unsafe transitions)
- **Application semantics:** collection pattern (`feed` and `entry`)

The semantics of OData are heavily inspired by the Atom Publishing Protocol. In fact, an OData API can serve Atom representations, and a client can treat an OData API as an AtomPub API with a whole lot of extensions. But I'll be considering OData as an API that serves mostly JSON representations.

Figure 10-5 shows a view of OData's protocol semantics, simplified to show only the parts of OData I'll be covering here. And here's an OData representation of a collection from a microblogging API, similar to Chapter 2's You Type It, We Post It:

```
{
  "odata.metadata":
    "http://api.example.com/YouTypeItWePostIt.svc/$metadata#Posts",
  "value": [
    {
      "Content": "This is the second post.",
      "Id": 2,
      "PostedAt": "2013-04-30T03:34:12.0992416-05:00",
      "PostedAt@odata.type": "Edm.DateTimeOffset",
      "PostedBy@odata.navigationLinkUrl": "Posts(2)/PostedBy",
      "odata.editLink": "Posts(2)",
      "odata.id": "http://api.example.com/YouTypeItWePostIt.svc/Posts(2)",
      "odata.type": "YouTypeItWePostIt.Post"
    },
    {
      "Content": "This is the first post",
      "Id": 1,
      "PostedAt": "2013-04-30T04:14:53.0992416-05:00",
      "PostedAt@odata.type": "Edm.DateTimeOffset",
      "PostedBy@odata.navigationLinkUrl": "Posts(1)/PostedBy",
      "odata.editLink": "Posts(1)",
      "odata.id": "http://api.example.com/YouTypeItWePostIt.svc/Posts(1)",
      "odata.type": "YouTypeItWePostIt.Post"
    },
  "#Posts.RandomPostForDate": {
    "title": "Get a random post for the given date",
    "target": "Posts/RandomPostForDate"
   }
}
```

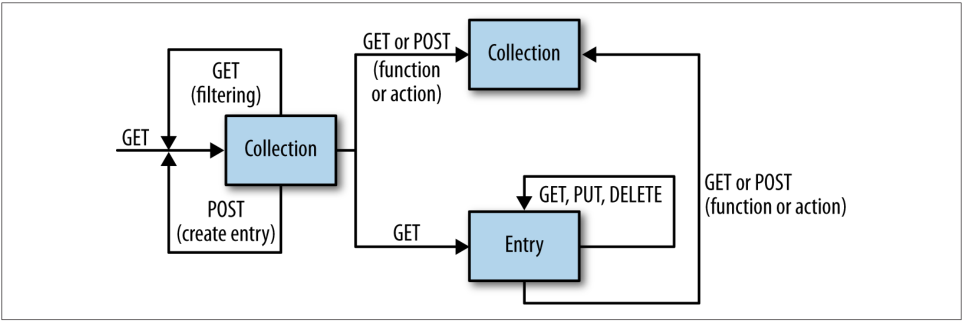


Figure 10-5. The protocol semantics of OData (simplified)

Like the other JSON-based formats we've seen, OData representations are JSON objects whose properties are named with short, mysterious strings. A property like `Content` or `PostedAt` is ordinary JSON data, and its name acts as a semantic descriptor. A property whose name includes the `odata.` prefix is a hypermedia control or some other bit of OData-specific metadata. Some examples from this document:

- The property `odata.id` contains a unique ID—that is, a URI—for one specific entry-type resource.
- The property `PostedAt@odata.type` contains semantic type information for the value of the `PostedAt` property. The type, `Edm.DateTimeOffset`, refers to OData's schema format: the Entity Data Model.
- The property `odata.editLink` acts like an AtomPub link with `rel="edit"`. If you want to modify or delete one of the example posts, you can send a PUT, PATCH, or DELETE request to the relative URL `Posts(2)` or `Posts(1)`.
- The property `PostedBy@odata.navigationLinkUrl` contains a hypermedia link to another resource. The application-specific part of the property name, `PostedBy`, serves as a link relation. In human terms, this is a link to the user who published this particular post.

The protocol semantics of OData resources repeat what you've already seen in Collection+JSON and AtomPub. A collection resource supports GET (to get a representation) and POST (to append a new entry to the collection). Entry-type resources support GET, as well as (via their `odata.editLink`) PUT, DELETE, and PATCH.

Filtering

OData also defines a set of implicit protocol semantics for filtering and sorting a collection, using a query language similar to SQL. If you know you have the URL to an OData collection, you can manipulate that URL in a wide variety of ways. Sending GET to the resulting URLs will yield representations that filter and paginate the collection in different ways.

I say these protocol semantics are implicit because you don't have to look for a hypermedia form that tells you how to make the HTTP request that carries out a particular search. You can construct that request based on rules found in the OData spec.

Let's look at a few examples. Suppose the (relative) base URL of the microblog collection is */Posts*. You don't need a hypermedia form to tell you how to search for blog posts that include the string "second" in their `Content` property. You can build the URL yourself[3]:

```
/Posts$filter=substringof('second', Content)
```

You can search for posts that include "second" in their `Content` and were `PostedBy` a resource whose `Username` property is "alice":

```
/Posts$filter=substringof('second', Content)+ and +PostedBy/Username eq 'alice')
```

You can pick up only the last five posts that were published in the year 2012:

```
/Posts$filter=year(PostedAt) eq 2012&$top=5
```

Want to get the second page of that list? You don't need to look for a link with the relation `next` in the representation. The URL you should use is defined by the OData spec:

```
/Posts$filter=year(PostedAt) eq 2012&$top=5&skip=5
```

By default, the microblog collection presents entries in reverse chronological order based on the value of the `PostedAt` property. If you want to use chronological order instead, the OData spec explains what URL you should use:

```
/Posts$orderBy=PostedAt asc
```

In the other collection-pattern standards, the server must serve a hypermedia control to explicitly describe each allowable family of searches. Collection+JSON serves search templates, AtomPub serves OpenSearch forms. An OData collection doesn't need to provide this information because every OData collection implicitly supports the entire OData query protocol. A client doesn't need a hypermedia form to know it's OK to send GET requests to certain URLs. The OData format itself puts additional constraints on the server that guarantee that certain URLs will work.

OData defines a few more bits of implicit protocol semantics, mostly pertaining to the relationships between resources. I won't be covering them here.

Functions and the metadata document

In addition to the impressive set of state transitions implicitly defined by OData's query protocol, an OData representation may include explicit hypermedia controls describing

3. All of these URLs need to be URL-encoded, obviously. I've left them unencoded for the sake of clarity.

any state transition at all. These controls have protocol semantics similar to HTML forms. Safe transitions are called "functions," and they use HTTP GET. Unsafe transitions are called "actions," and they use HTTP POST. I'll be focusing on functions, but actions work the same way.

Here's a simple OData form that takes a date as input. It triggers a state transition where the server looks at all of a microblog's entries from the given date, picks one at random, and serves a representation of it.

```
"#Posts.RandomPostForDate": {
    "title": "Get a random post for the given date",
    "target": "Posts/RandomPostForDate"
  },
```

If this was a simple query like "all the microblog entries from a given date," the form wouldn't be necessary. The state transition would be implicitly described by OData's query protocol. But that protocol can't express the concept of "random selection," so this state transition must be described explicitly, using a hypermedia form. Now, here's a question: can you look at this form and figure out which HTTP request to make?

It's a trick question. You can't figure it out, because I didn't show you the whole form. The part of the form gives you the base URL to use (*Posts/RandomPostforDate*), but it doesn't explain how to format your contribution—the date for which you want a random post. It's equivalent to this HTML form:

```
<form action="Posts/RandomPostForDate" method="GET">
 <input class="RandomPostForDate" type="submit"
  value="Get a random post for the given date."/>
</form>
```

That's obviously incomplete. It's missing a formal description for "the given date." What format should "the given date" take? What's its semantic descriptor? Do you trigger the state transition by sending GET to *Posts/RandomPostforDate?Date=9/13/2009*, or to *Posts/RandomPostForDate?the_date_to_use=13%20August%202009*, or to *Posts/RandomPostForDate?when=yesterday*? You just don't have that information.

In the HTML example, the missing information should go into a second `<input>` tag within the `<form>` tag. But with OData, that information is kept in a different document —a "metadata document" written not in JSON but in XML, using a vocabulary called the Comma Schema Definition Language (CSDL).[4]

An OData representation links to its metadata document using the `odata.metadata` `property`

```
{
  "odata.metadata":
```

4. For more information on CSDL, go to the OData website.

```
        "http://api.example.com/YouTypeItWePostIt.svc/$metadata#Posts",
      ...
    }
```

Here's the part of the metadata document that completes the definition of the `RandomPostForDate` state transition:

```
<FunctionImport Name="RandomPostforDate" EntitySet="Posts"
                IsBindable="true" m:IsAlwaysBindable="false"
                ReturnType="Post" IsSideEffecting="false">
 <Parameter Name="date" Type="Edm.DateTime" Mode="In" />
</FunctionImport>
```

Now you know the whole story. You trigger the state transition `RandomForDate` by formatting a date as a string, in a format defined by OData's Entity Data Model.[5] You know that this state transition is safe, because its CSDL description has the `IsSideEffecting` attribute set to `false`. That means you should trigger the state transition with a GET request rather than with POST.

Combine the metadata document with the OData representation, and you have all the information necessary to trigger the state transition `RandomPostForDate`. You send an HTTP request that looks something like this:

```
GET /YouTypeItWePostIt.svc/Posts/RandomPostForDate?date=datetime'2009-08-13T12:↵
00' HTTP/1.1
Host: api.example.com
```

Although `RandomPostForDate` is a simple transition, OData state transitions can get very complicated. The metadata document stores the messy details that explain exactly how to trigger whatever state transitions you might find mentioned in an OData document. This saves the server from having to include a complete description of a complex state transition in every representation that supports it. A client that's interested in a given state transition can look up a complete description of it.

Metadata documents as service description documents

I've presented OData in a way that makes it look like Collection+JSON or Siren. A microblog post is represented as a JSON object containing data fields like `DatePublished`, along with hypermedia controls and other "metadata" explaining the possible next steps.

That's the version of OData I recommend, and it has the media type `application/json;odata=fullmetadata`. But there's another way to write down an OData document: a way that keeps *all* the hypermedia controls, not just the complicated ones, in the metadata document.

5. The EDM is defined in the same document as CSDL.

The media type of such a document is application/json;odata=minimalmetadata. Here's what a representation of the microblog would look like in this format:

```
{
  "odata.metadata":
    "http://api.example.com/YouTypeItWePostIt.svc/$metadata#Posts",
  "value": [
    {
      "Content": "This is the first post.",
      "Id": 1,
      "PostedAt": "2013-04-30T01:42:57.0901805-05:00"
    },
    {
      "Content": "This is the second post.",
      "Id": 2,
      "PostedAt": "2013-04-30T01:45:03.0901805-05:00"
    },
  ]
}
```

That's a lot smaller, but in the world of REST, smaller isn't necessarily better. Where'd the metadata go? What happened to PostedBy@odata.navigationLinkUrl and #Posts.RandomPostForDate? How are you supposed to decide which HTTP request to make next?

All of that information went into the CSDL document at the other end of the odata.metadata link. I showed you part of the CSDL document earlier when I was discussing RandomPostForDate, but here's a bit more of it (this excerpt shows what happened to PostedBy and RandomPostForDate):

```
<edmx:Edmx Version="1.0"
 xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
 <edmx:DataServices
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
  m:DataServiceVersion="3.0" m:MaxDataServiceVersion="3.0">

  <Schema Namespace="YouTypeItWePostIt">
   <EntityType Name="Post">
    <Key><PropertyRef Name="Id"/></Key>
    <Property Name="Id" Type="Edm.Int32" Nullable="false"/>
    <Property Name="Content" Type="Edm.String"/>
    <Property Name="PostedAt" Type="Edm.DateTimeOffset" Nullable="false"/>
    <NavigationProperty Name="PostedBy"
     Relationship="YouTypeItWePostIt.Post_PostedBy"
     ToRole="PostedBy" FromRole="Post"/>
   </EntityType>

   ...

   <EntityContainer Name="YouTypeItWePostItContext"
    m:IsDefaultEntityContainer="true">
    <EntitySet Name="Posts" EntityType="YouTypeItWePostIt.Post"/>

    <FunctionImport Name="RandomPostforDate" EntitySet="Posts"
                    IsBindable="true" m:IsAlwaysBindable="false"
                    ReturnType="Post" IsSideEffecting="false">
     <Parameter Name="date" Type="Edm.DateTime" Mode="In" />
    </FunctionImport>

    <EntitySet Name="Users" EntityType="YouTypeItWePostIt.User"/>

   </EntityContainer>

  ...

  </Schema>
 </edmx:DataServices>
</edmx:Edmx>
```

There's nothing wrong with keeping extra information about a resource outside of that resource's representation. After all, that's what a profile or a JSON-LD context does. The problem here is that the CSDL document can be seen as a service description document: an overview of the API as a whole that makes it look like a relational database.

As I mentioned in Chapter 9, users who see a document like this have a tendency to automatically generate client code based on it. Doing this creates a tight coupling between the generated client and this specific edition of the service description. If the server implementation changes, the CSDL document will change along with it, but the clients won't change to match. They'll just break.

Fortunately, no one is making you use OData this way. If you use the media type `application/json;odata=fullmetadata`, your OData representations will contain their own hypermedia controls. A client will only need to consult the CSDL metadata document when it needs to trigger a complicated state transition—a function or action—that can't be completely described with OData.

Pure Hypermedia Formats

These media types have very generic application semantics, or else they have no application semantics at all. They focus on representing the protocol semantics of HTTP. You provide your own application semantics, by plugging link relations and semantic descriptors into predefined slots.

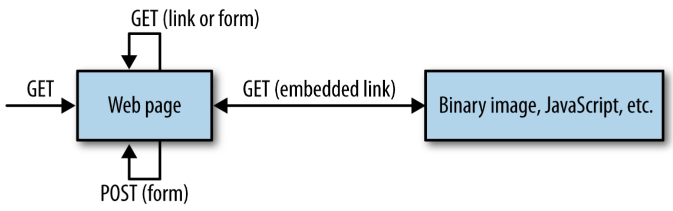
HTML

- **Media types:** `text/html` and `application/xhtml+xml`
- **Defined in:** open standards for HTML 4, for XHTML, and for HTML 5

- **Medium:** XML-like
- **Protocol semantics:** navigation through GET links; arbitrary state transitions through forms (GET for safe transitions, POST for unsafe transitions)
- **Application semantics:** human-readable documents ("paragraph," "list," "table," "section," etc.)
- **Covered in:** Chapter 7

The original hypermedia format, and a highly underrated choice for an API. HTML can make direct use of microformats and microdata, instead of using an approximation such as an ALPS profile. HTML's `<script>` tag lets you embed executable code to be run on the client, a feature of RESTful architectures ("code on demand"; see Appendix C) not supported by any other hypermedia format. And HTML documents can be graphically displayed to human beings—invaluable for APIs designed to be consumed by an Ajax or mobile client, and useful when debugging any kind of API.

Here's HTML's state diagram:



HTML comes in three flavors. HTML 4 has been the stable standard since 1997. HTML 5, its replacement, is still under development. There's also XHTML, an HTML-like format that happens to be valid XML.

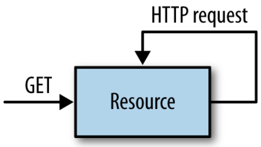
As far as this book is concerned, the only important differences between these three standards are HTML 5's new rules for client-side input validation, and the fact that HTML 5 will eventually support microdata.

HAL

- **Media types:** `application/hal+json` and `application/hal+xml`
- **Defined in:** the JSON version is defined in the Internet-Draft "draft-kelly-json-hal"; the XML version is defined in a [personal standard here]
- **Medium:** Either XML or JSON
- **Protocol semantics:** arbitrary state transitions through links that may use any HTTP method; links do not mention the HTTP method to be used—that's kept in human-readable documentation
- **Application semantics:** none to speak of

- **Covered in:** Chapter 7

HAL is a minimalist format. Its state diagram is so generic it looks like something out of the HTTP specification:



HAL relies on custom link relations (and their human-readable explanations in profiles) to do the heavy lifting.

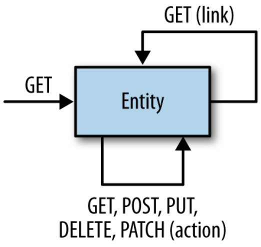
Siren

- **Media types:** `application/vnd.siren+json`
- **Defined in:** personal standard
- **Medium:** JSON (an XML version is planned)
- **Protocol semantics:** navigation through GET links; arbitrary state transitions through "actions" (GET for safe actions, POST/PUT/DELETE for unsafe actions)
- **Application semantics:** very generic

A Siren document describes an "entity," a JSON object that has approximately the same semantics as HTML's `<div>` tag. An entity may have a "class" and a list of "properties." It may contain a list of "links," which work like HTML `<a>` tags (with a `rel` and an `href`). It may also contain a list of `actions`, which work like HTML `<form>` tags (with a `name`, an `href`, a `method`, and a number of `fields`).

An entity may also have some number of subentities, similar to how one `<div>` tag may contain another. You can implement the collection pattern this way.

Siren's state diagram looks like a cross between HAL's and HTML's:



The Link Header

- **Media type:** n/a
- **Described in:** RFC 5988
- **Medium:** HTTP header
- **Protocol semantics:** navigation through GET links
- **Application semantics:** none
- **Covered in:** Chapter 4

The `Link` header is not a document format, but I'm putting it in the zoo because it lets you add simple GET links to representations that lack hypermedia controls, like binary images or JSON documents. The header's `rel` parameter is a slot for the link relation:

```
Link: <http://www.example.com/story/part2>;rel="next"
```

RFC 5988 defines some other useful parameters for the `Link` header, including `type` (which gives a hint as to the media type at the other end of the link) and `title` (which contains a human-readable title for the link).

As far as I'm concerned, the most important use of the `Link` header is to connect a JSON document with a profile. JSON is incredibly popular despite having no hypermedia controls, and the `application/json` media type doesn't support the `profile` parameter, so `Link` is the only reliable way to point to the profile that explains what a JSON document means.

```
Content-Type: application/json
Link: <http://www.example.com/profiles/hydraulics>;rel="profile"
```

The Location and Content-Location Headers

- **Media type:** n/a
- **Described in:** RFC 2616
- **Medium:** HTTP header
- **Protocol semantics:** depends on the HTTP response code
- **Application semantics:** none
- **Covered in:** Chapter 1, Chapter 2, Chapter 3, Appendix B

Here are two simple hypermedia controls defined in the HTTP standard itself. I've mentioned `Location` in passing, but I'll give both detailed coverage in Appendix B.

The `Content-Location` header points to the canonical location of the current resource. It's equivalent to a link that uses the IANA-registered link relation `canonical`.

The `Location` header is used as an all-purpose link whenever the protocol semantics of an HTTP response demand a link. The exact behavior depends on the HTTP status code. When the response code is 201 (`Created`), the `Location` header points to a newly created resource. But when the response code is 301 (`Moved Permanently`), the `Location` header points to the new URL of a resource that moved. Again, the details are in Appendix B.

URL Lists

- **Media type:** `text/uri-list`
- **Medium:** none
- **Described in:** RFC 2483
- **Protocol semantics:** none
- **Application semantics:** none

A `text/uri-list` document is just a list of URLs:

```
http://example.org/
https://www.example.com/
...
```

This is probably the most basic hypermedia type ever devised. It doesn't support link relations, so there's no way to express the relationship between these URLs and the resource that served the list. There are no explicit hypermedia controls, so the client has no way of knowing what kind of requests it's allowed to send to these URLs. The best you can do is send a GET request to each and see what kind of representations you get.

JSON Home Documents

- **Media type:** `application/json-home`
- **Described in:** Internet-Draft "draft-nottingham-json-home"
- **Medium:** JSON
- **Protocol semantics:** completely generic
- **Application semantics:** none

JSON Home Documents are a more sophisticated version of URL lists. The format is intended for use as the "home page" of an API, listing all the resources provided and their behavior under the HTTP protocol.

A JSON Home Document is a JSON object. The keys are link relations, and the values are JSON objects known as "Resource Objects." Here's an example from the world of the maze game:

```
{
 "east": { "href": "/cells/N" },
 "west": { "href": "/cells/L" }
}
```

A Resource Object is a hypermedia control that describes the protocol semantics of a resource, or a group of related resources. Here's a search form, described by a URI Template:

```
{
 "search": {"href-template": "/search{?query}",
            "href-vars": {
              "query" : "http://alps.io/opensearch#searchTerms"
             }
}
```

A Resource Object may include "resource hints" that describe its protocol semantics in more detail. The most common hint is `allow`, which explains which HTTP methods the resource will respond to. Here's a JSON Home Document that uses the `flip` link relation I defined for my extension of the maze game:

```
{
 "flip": { "href": "/switches/4",
           "hints": { "allow": ["POST"] }
         }
}
```

A JSON Home Document says nothing about the application semantics of the resources it links to. That information is kept in the representations on the other side of the links.

By combining a JSON Home Document (which describes an API's protocol semantics) with an ALPS document (which describes its application semantics), you can take an existing API—even one that doesn't use hypermedia—and move most of its human-readable documentation into a structured, machine-readable format.

The Link-Template Header

- **Media type:** n/a
- **Described in:** Internet-Draft "draft-nottingham-link-template" (see also RFC 6570)
- **Medium:** HTTP header
- **Protocol semantics:** navigation through GET
- **Application semantics:** none

The `Link-Template` header works exactly the same way as the `Link` header, except its value is interpreted as a URI Template (RFC 6570) instead of as a URL. Here's a search form in an HTTP header:

```
Link-Template: </search{?family-name}>; rel="search"
```

The `Link-Template` header has a special variable called `var-base`, which allows you to specify a profile for the variables in the URI Template. In the example, the variable name `family-name` is suggestive of what kind of value you should plug into the variable, but it doesn't technically mean anything. It might as well be called `put-something-here.` Add a `var-base`, and suddenly there's a link to a formal definition of `family-name`.

```
Link-Template: </search{?family-name}>; rel="search";↵
var-base="http://alps.io/microformats/hCard#"
```

Now the variable `family-name` expands to the URL *http://alps.io/microformats/hCard#family-name*. The ALPS document at the other end of that URL explains the application semantics of the `family-name` variable.

Here's another example that uses schema.org's application semantics instead of ALPS:

```
Link-Template: </search{?familyName}>; rel="search"; var-base="http://schema.org/"
```

Here, the variable `familyName` expands to the URL *http://schema.org/familyName*, which means basically the same thing as *http://alps.io/microformats/hCard#family-name*.

As of this writing, the Internet-Draft defining the `Link-Template` header has expired. The author of the draft, Mark Nottingham, told me to go ahead and put it in the book anyway. He said he'll revive the Internet-Draft if more people become interested in `Link-Template`.

WADL

- **Media type:** `application/vnd.sun.wadl+xml`
- **Defined in:** open standard
- **Medium:** XML
- **Protocol semantics:** completely generic
- **Application semantics:** none, minimal support for extensions

WADL was the first hypermedia format to support a complete set of protocol semantics. A WADL `<request>` tag (analogous to an HTML form) can describe an HTTP request that uses any method, provides values for any specified HTTP request headers, and includes an entity-body of any media type. Like AtomPub, this doesn't sound very special now, but it was groundbreaking at the time. WADL can describe the protocol semantics of *any* web API, even one that's poorly designed and violates the HTTP standard.

Here's a snippet of WADL that explains how to flip a switch in Chapter 7's version of the maze game:

```
<method id="flip" name="POST" href="/switches/4">
 <doc>Flip the switch</doc>
</method>
```

WADL can also describe the *content* of XML representations. A WADL document can point out which parts of a representation are interesting—notably, which parts are links to other resources. A WADL document can bring in an XML Schema document to explain the data types of the XML data it describes. This is useful when an XML representation has no associated schema of its own.

WADL's `<doc>` tag makes it a basic profile format, capable of describing the application semantics of an HTTP request or the inside of an XML representation. But WADL can't describe the inside of a JSON representation at all.[6]

WADL is not in widespread use, but there are some Java JAX-RS implementations that generate WADL descriptions of APIs. Therein lies the problem. An automatically generated description of an API is likely to be tightly coupled to the server-side implementation. What's more, an API that uses WADL typically serves one enormous WADL document describing the protocol semantics of the entire API.

This is a service description document, and as I mentioned in Chapter 9 , it encourages users to create automatically generated clients, based on the assumption that they've obtained a complete and unchanging overview of the API's semantics.

But APIs change. When that happens, the WADL description of the API will also change, but the automatically generated clients will not. The clients will break.

XLink

- **Media type:** n/a
- **Defined in:** W3C standard
- **Medium:** XML documents
- **Protocol semantics:** navigation and transclusion with GET
- **Application semantics:** none

XLink is a plug-in standard that lets you add hypermedia links to any XML document. Unlike HTML and Maze+XML, XLink doesn't define special XML tags that represent hypermedia links. XLink defines a family of attributes that can be applied to *any* XML tag to turn that tag into a link.

Here's an ad hoc XML representation of a cell in the maze game. The `<root>` and `<direction>` tags are tag names I made up for demonstration purposes—they have no

6. The JSON Pointer standard, defined in the Internet-Draft appsawg-json-pointer, may fix this.

hypermedia capabilities of their own, but I can turn them into links by adding XLink attributes.

```
<?xml version="1.0"?>
<root xmlns:xlink="http://www.w3.org/1999/xlink">
  <direction
    xlink:href="http://maze-server.com/maze/cell/N"
    xlink:title="Go east!"
    xlink:arcrole="http://alps.io/example/maze/#east"
    xlink:show="replace"
  />

<link
    xlink:href="http://maze-server.com/maze/cell/L"
    xlink:title="Go west!"
    xlink:arcrole="http://alps.io/example/maze/#west"
    xlink:show="replace"
  />
</root>
```

The `href` and `title` attributes should look familiar. The link relation goes into the optional `arcrole` attribute. There's a slight twist here: the `arcrole` attribute only supports extension link relations—the ones that look like URLs. Your link relation can't look like `author` or `east`; it has to look like `http://alps.io/maze/#west`.

The `show` attribute lets you switch between a navigation link that works like HTML's `<a>` tag (`show="replace"`, the default) and an embedding link that works like HTML's `<img>` tag (`show="embed"`). The HTTP method used is always GET.

With XLink, I can give an ad hoc XML vocabulary approximately the same hypermedia capabilities that were designed into Maze+XML. There are a few advanced features of XLink I haven't covered: notably, the extended link type, which lets you connect more than two resources using a single link, and the `role` attribute, which I'll show off in Chapter 12.

XForms

- **Media type:** `n/a`
- **Medium:** XML documents.
- **Protocol semantics:** arbitrary state transitions through forms (GET for safe transitions, POST/PUT/DELETE for unsafe transitions)
- **Application semantics:** none

XForms does for hypermedia forms what XLink does for links. It's a plug-in standard that adds HTML-like forms to any XML document. Unlike XLink, though, it does define its own tags. Here's how XForms might represent a simple search form:

```
<xforms:model>
 <xforms:submission action="http://example.com/search" method="get"
                    id="submit-button"/>
 <xforms:instance>
  <query/>
 </xforms:instance>
<xforms:model>
```

The `<model>` tag is a container, like HTML's `<form>` tag. The `<submission>` tag explains what HTTP request to make: in this case, a GET request to *http://example.com/search*. The children of the `<instance>` tag explain how to construct the query string (for a GET request) or the entity-body (for a POST or PUT request).

The `<query>` tag is one I made up for this example; it represents a form field called `query`. The meaning of this tag—e.g., whether it's a text field or a checkbox—is defined separately, in an XForms `<input>` tag:

```
<xforms:input ref="query">
 <xforms:label>Search terms</xforms>
</xforms:input>

<xforms:submit submission="submit-button">
 <label>Search!</label>
</xforms:submit>
```

The `<input>` tag with `ref="query"` says that the `query` field is a text input with a human-readable `<label>`. The `<submit>` tag gives a `<label>` to the submit button. Together, the `<model>` tag and the two `<input>` tags approximate the functionality of this HTML form:

```
<form action="http://example.com/search" method="GET">
 <input type="text" name="query"/>
 <label for="query">Search terms</label>
 <submit value="Search!">
</form>
```

This is a very basic example; there are many advanced features of XForms that I won't be covering. The W3C's tutorial "XForms for XHTML Authors"[7] uses HTML forms to explain XForms in some detail, going beyond the capabilities of pure HTML into some of the advanced features of XForms.

GeoJSON: A Troubled Type

We've seen the healthy specimens in the hypermedia zoo. Now I'd like to take a look at GeoJSON, a domain-specific document format with some design flaws that hurt its

7. The tutorial is available at this w3.org page.

usability in APIs.[8] I'm not doing this to pick on GeoJSON; I've made exactly the same mistakes myself. They're common mistakes, so even if GeoJSON doesn't sound like something you need to learn about right now, stick around.

GeoJSON is a standard based on JSON, designed for representing geographic features like points on a map. Here are its stats:

- **Media type:** `application/json`
- **Defined in:** corporate standard defined here
- **Medium:** JSON
- **Protocol semantics:** GET for transclusion of coordinate systems
- **Application semantics:** geographic features and collections of features

Like almost all JSON-based documents used in APIs, a GeoJSON document is a JSON object that must contain certain properties. Here's a GeoJSON document that pinpoints the location of an ancient monument on Earth:

```
{
 "type": "FeatureCollection",
 "features":
 [
  {
   "type": "Feature",

   "geometry":
   {
    "type": "Point",
    "coordinates": [12.484281,41.895797]
   },

   "properties":
   {
    "type": null,
    "title": "Column of Trajan",
    "awmc_id": "91644",
    "awmc_link": "http://awmc.unc.edu/api/omnia/91644",
    "pid": "423025",
    "pleiades_link": "http://pleiades.stoa.org/places/423025",
    "description": "Monument to the emperor Marcus Ulpius Traianus"}
  }
 ]
}
```

8. These flaws don't hurt GeoJSON so much that no one uses it. It's pretty popular—just not as good as it could be.

I adapted this representation slightly from the real-world API provided by UNC's Ancient World Mapping Center. GeoJSON's application semantics are simple, and it should be fairly easy for a human to understand the document. It represents a collection called a `FeatureCollection`. The collection only contains one item: a `Feature`, which has a `geometry` (a single `Point` on the map) and a bunch of miscellaneous `properties` like the human-readable `description`.

A quick look at the GeoJSON standard reveals that instead of a `Point`, the `geometry` could have been a `LineString` (representing a border or a road) or a `Polygon` (representing the area of a city or country).

GeoJSON Has No Generic Hypermedia Controls

Unfortunately, GeoJSON's protocol semantics are anything but straightforward. Do you see `awmc_link` and `pleiades_link` in that representation? They look like hypermedia links, but they're not. According to the GeoJSON standard, those are just strings that happen to look like URLs. When the Ancient World Mapping Center designed their GeoJSON API, they had to stuff all their links into the `properties` list, because GeoJSON doesn't define hypermedia controls for them. This means a generic GeoJSON client can't follow the `pleiades_link`, or even recognize it as a link. To follow that link, you'll need to write a client specifically for the Ancient World Mapping Center's API.

If GeoJSON didn't define any hypermedia controls, this would be understandable. Not every data format has to be a hypermedia format. I simply wouldn't mention GeoJSON in this book. The odd thing is that GeoJSON *does* define a hypermedia control, but it can only be used for one specific thing: changing the coordinate system in use.

By default, the coordinates in a GeoJSON representation (`[12.484281,41.895797]`) are measured in degrees of longitude and latitude—a system we're all familiar with. Since the planet Earth is not a perfect sphere, these measurements are interpreted according to a standard called WGS84,[9] which lays down things like the approximate shape of Earth, the location of the prime meridian, and what "sea level" means.

If you're not a map geek, you can assume Earth is a sphere and be done with it. But for map geeks, WGS84 is just a default. There are many other coordinate systems you could use. British readers may be familiar with the Ordnance Survey National Grid, a coordinate system that uses "easting" and "northing" instead of latitude and longitude, and that can only represent points within a specific 700-by-1300-kilometer area that covers the British Isles. There are *infinitely many* coordinate systems, since you can define a system that puts Earth's prime meridian wherever you want.

9. An industry standard, but from a different industry than the rest of the standards mentioned in this book. You can get a PDF version of the standard at this page.

And now our story comes back to hypermedia, because this is what GeoJSON's sole hypermedia control is for. GeoJSON lets you link to a description of the coordinate system you're using.

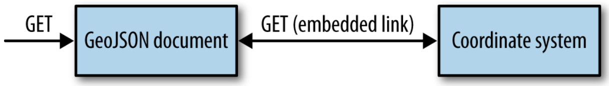
Here's a GeoJSON document containing a genuine hypermedia link that any GeoJSON client will recognize as such:

```
{
 "type":"Feature",
 "geometry":
 {
  "type":"Point",
  "coordinates":[60000,70000]
 },

 "crs": {
  "type": "link",
  "properties": {
    "href": "http://example.org/mygrid.wkt",
    "type": "esriwkt"
    }
  }
}
```

The coordinates `[60000,70000]` are not valid measurements of longitude and latitude, but that's fine, because we're not using longitude and latitude. We're using a custom coordinate reference system (`crs`) described by the resource at *http://example.org/mygrid.wkt*. This is exactly the sort of thing hypermedia is good for. The problem with GeoJSON is that the *only* place it allows a link is within the definition of a coordinate reference system.

This state diagram describes GeoJSON's protocol semantics:



That's not very useful! Most GeoJSON APIs don't use custom coordinate systems—we're all used to ordinary longitude and latitude. But the GeoJSON standard allows for them, because they are an essential aspect of the problem domain. On the other hand, pretty much any API needs to serve miscellaneous links between its resources, but the GeoJSON standard lacks that capability, presumably because it's *not* directly related to the problem domain. The underlying data format is no help, since JSON defines no hypermedia controls at all. That's why API implementers must resort to hacks like `awmc_link`.

Enough complaining; what would I do differently? A design more focused on hypermedia would allow a list of `links`, each of which could specify a link relation. GeoJSON would look a lot more like Collection+JSON or Siren. Then the Ancient World Mapping

Center wouldn't need to smuggle awmc_link and pleiades_link into the properties object.

To link to a coordinate system, you'd use the same kind of link you'd use for anything else. GeoJSON's crs would become a link relation, useful in any mapping application, even one that doesn't use GeoJSON.

It's OK to have application-specific hypermedia controls. HTML's <img> tag is an application-specific hypermedia control. But you also need to make available a simple, generic link control.

GeoJSON Has No Media Type

There's another problem with GeoJSON: it has no registered media type. A GeoJSON document is served as application/json, just like any other JSON document. How is a client supposed to distinguish between GeoJSON and plain old JSON?

The best solution is for the server to treat GeoJSON as a profile of JSON. This means serving a link to the GeoJSON standard with rel="profile". Since JSON on its own has no hypermedia controls, you'll need to use the Link header:

```
Link: <http://www.geojson.org/geojson-spec.html>;rel="profile"
```

You could also write an ALPS profile or JSON-LD context for GeoJSON, and serve a link to that using the Link header:

```
Link: <http://example.com/geojson.jsonld>;↵
rel="http://www.w3.org/ns/json-ld#context"
```

As far as I know, there's no GeoJSON implementation that does either of these. GeoJSON is served as application/json and the client is simply expected to know ahead of time which resources serve GeoJSON representations and which serve ordinary JSON. A client that wants to understand different profiles of JSON must run heuristics against every incoming JSON representation, trying to figure out which profile the server is giving it.

Does it sound unrealistic that one client would need to handle different profiles of JSON? Well, consider this. The ArcGIS platform includes an API that presents the same kind of information as GeoJSON. It serves JSON representations that superficially resemble GeoJSON's representations, and it serves them as application/json, with no profile information.

I don't think it's a ludicrous fantasy to imagine a client that can handle both GeoJSON and ArcGIS JSON. If GeoJSON was served as application/geo+json and ArcGIS JSON was served as application/vnd.arcgis.api+json, a client developer could split up the client code based on the value of the Content-Type header, and reunite the code paths once the incoming data was parsed. If GeoJSON and ArcGIS JSON were consistently served as different profiles, a developer could split up the code based on the value of the `Link` header. If they were served with different JSON-LD contexts, a developer could split up the code based on that.

But both formats are served as though they meant the same thing. A unified client must try to distinguish between the two formats using poorly defined heuristics. Or, more likely, the idea of a unified client never occurs to anyone. Like two ships passing in the night, one developer writes a GeoJSON client for GeoJSON APIs, while another duplicates much of the first developer's work, writing an ArcGIS client to run against ArcGIS installations.

No one is to blame for this. The GeoJSON standard was finalized in 2008. Back then, our understanding of hypermedia APIs was pretty poor. The GeoJSON designers didn't forget to register a media type; they considered it and then tabled the issue.

But it's not 2008 anymore. We now have standards that add real hypermedia controls to JSON. We can use profiles to add application-level semantics to generic hypermedia types. We've seen hundreds of one-off, mutually incompatible data formats served as `application/json`, and we know we can do better.

Learning from GeoJSON

When a GeoJSON object is included in a hypermedia-capable JSON document (such as an OData document, which has explicit support for embedded GeoJSON), both of these problems go away. It doesn't matter that GeoJSON has no general hypermedia controls, because it's embedded in a document that can take care of that stuff. It doesn't matter that GeoJSON has no special media type, because it inherits the media type of the parent document. At this point, GeoJSON becomes a plug-in standard, similar to OpenSearch.

If you design a domain-specific format that's not clearly a plug-in for some other format, you should give it a unique media type. It helps if you also register the media type with the IANA, but if you use the `vnd.` prefix, you don't have to register anything.

Also make sure your format features some kind of general hypermedia control, like Maze+XML's `<link>` tag. You might think it's not your job to provide a generic hypermedia control, since that has nothing to do with your problem domain. But if you don't provide a hypermedia control, every one of your users will come up with their own one-off design, a la *awmc_link*. You may be able to borrow a simple clip-on hypermedia control by adopting XLink for XML documents, or JSON-LD for JSON documents.

All in all, it may be better to forget the domain-specific media type, and design a domain-specific set of application semantics—a profile. Those semantics can then be plugged in to a general hypermedia type like Siren, or a collection-pattern media type like Collection+JSON.

The Semantic Zoo

I've shown you the wonders of the hypermedia zoo to demonstrate the diversity and flexibility of hypermedia-based designs. Now I'm going to take you on a (much quicker) tour of a different zoo: a series of butterfly gardens full of application semantics for different problem domains. My goal here is more concrete: to help you save time by reusing work other people have already done.

In Chapter 9, I played up the benefits of reusing existing application semantics. The profiles listed here are the result of smart people carefully considering a problem domain and navigating tricky naming issues. There's no reason you should have to duplicate that work. Reusing existing semantics whenever possible also removes the temptation to expose your server's implementation details, leaving you free to change those details without hurting your clients.

Most important of all, when different APIs share the same application semantics, it becomes possible to write interoperable clients, or general semantics-processing libraries, instead of a custom client for each individual API. This is more of a hope than a reality right now, but at least the immediate path forward is clear.

Rather than show you a lot of individual profiles in the semantic zoo, I'll focus mainly on the registries that house the profiles.

The IANA Registry of Link Relations

- **Media types:** any
- **Site:** this IANA page
- **Semantics:** general navigation

I've talked about the IANA registry of link relations for practically the entire book. It's a global registry containing about 60 link relations. You're allowed to use any IANA-registered relation in any representation, and to assume that your clients know what you're talking about.

Link relations only make it into the IANA registry if they are defined in an open standard such as an RFC or W3C Recommendation, and are generic enough to be useful for any media type. Each link relation is given a short human-readable description and a link to the standard that originally defined it.

In step 3 of Chapter 9's design procedure, I mention several IANA-registered link relations that are especially useful for API design.

The Microformats Wiki

- **Media types:** HTML (ALPS versions are available for some microformats)

- **Site:** this microformats page
- **Semantics:** the kind of things a human being might want to search for online

The Microformats project was the first successful attempt at defining profiles for application semantics. Microformats are defined collaboratively, on a wiki and mailing list. Of the stable microformats, these are the ones you're most likely to be interested in:

hCalendar
: Describes events in time. Based on the plain-text iCalendar format defined in RFC 2445.

hCard
: describes people and organizations. Based on the plain-text vCard format (defined in RFC 2426), and covered in Chapter 7.

XFN
: A set of link relations describing relationships between people, ranging from `friend` to `colleague` to `sweetheart`.

XOXO
: Describes outlines. This microformat is interesting because it doesn't add anything to HTML at all. It just suggests best practices for using HTML's existing application semantics.

These microformat specifications are technically drafts, but most of them haven't changed in several years, so I'd say they're pretty stable:

adr
: Physical addresses. This is a subformat of hCard, including only the parts that represent addresses. The idea is that if you don't need all of hCard, you can just use adr.

geo
: Latitude and longitude. (Using the WGS84 standard, naturally!) Another subformat of hCard.

hAtom
: Blog posts. Based on the Atom feed format (RFC 4287). This is an interesting example of one hypermedia format (HTML) adopting the application semantics of another (Atom).

hListing
: Listings of services for hire, personal ads, and so on. This microformat mostly reuses semantics from related microformats: hReview, hCard, and hCalendar.

hMedia
: Basic metadata about image, video, and audio files.

hNews
: An extension of hAtom that adds a few extra descriptors specific to news articles, like `dateline`.

hProduct
: Product listings.

hRecipe
: Recipes.

hResume
: Resumes/CVs.

hReview
: Describes a review (of anything), with a `rating`.

There are several interesting microformats I haven't mentioned because they were effectively adopted by HTML 5, and are now IANA-registered link relations: `author`, `nofollow`, `tag`, and `license`. The rel-payment microformat also became the IANA-registered link relation `payment`.

I've created ALPS documents that capture the essential application semantics of most of the microformats listed here. They are available from the ALPS registry.

Link Relations from the Microformats Wiki

- **Media types:** HTML
- **Site:** this microformats page
- **Semantics:** very, very miscellaneous

The Microformats wiki also has a huge list of link relations defined in standards or seen in real usage, but not registered with the IANA. This wiki page is the official registry for link relations used in HTML 5, but it's also an unofficial registry of *all* link relations that aspire to be useful outside a single application. Maze+XML's link relations would never cut it with the IANA—they're too application-specific—but they're mentioned on the Microformats wiki.

In Chapter 8, I mentioned this wiki page and gave some examples of the relations defined there. I don't recommend simply picking up link relations from this wiki page and using them. Your clients will have no idea what you're talking about. The real advantage of this page is as a way of finding standards you didn't know about before.

If you were planning on making your own maze game API, and you searched this page for `maze` or `north`, you'd discover Maze+XML. You wouldn't necessarily end up *using*

Maze+XML, but you'd have a glimpse into how someone else had solved a similar problem.

schema.org

- **Medium:** HTML5, and RDFa (ALPS versions are available)
- **Site:** schema home page
- **Semantics:** the kind of things a human being might want to search for online

As I mentioned in Chapter 8, the main source for microdata items is a clearinghouse called schema.org. This site takes the application semantics of standards like rNews (for news) and GoodRelations (for online stores) and ports them to microdata items. In turn, I've automatically generated ALPS documents for schema.org's microdata items and made them available from alps.io.

There are hundreds of microdata items described on schema.org, and more are on the way as the schema.org maintainers work with the creators of other standards to represent those standards in microdata. Rather than talk about all of the microdata items, I'll list the current top-level items and mention some of their notable subclasses:

- CreativeWork (including Article, Blog, Book, Comment, MusicRecording, SoftwareApplication, TVSeries, and WebPage)
- Event (including BusinessEvent, Festival, and UserInteraction)
- Intangible is sort of a catch-all category, which notably includes Audience, Brand, GeoCoordinates, JobPosting, Language, Offer, and Quantity
- MedicalEntity (including MedicalCondition, MedicalTest, and AnatomicalStructure)
- Organization (including Corporation, NGO, and SportsTeam)
- Person
- Place (including City, Mountain, and TouristAttraction)
- Product (including ProductModel)

As you can see, there's a lot of overlap between schema.org microdata items and the microformats. The Person item covers the same ground as the hCard microformat. The Event item is similar to hEvent, Article to hAtom, NewsArticle to hNews, Recipe to hRecipe, GeoCoordinates to geo, and so on.

A word of caution: the schema.org microdata items are very consumer-focused. A Product is something the client can buy, not a project the client is working on. The semantics of the Restaurant item have a lot to do with eating at a restaurant, and almost nothing to do with running one or inspecting one. There's a SoftwareApplication item, but nothing for a bug, a unit test, a version control repository, a release milestone, or any of the other things we deal with when we *develop* software. To my eyes, the only item described in enough detail to be useful to a practitioner is MedicalEntity, and a doctor would probably disagree with me on that.

In short, the schema.org project has a definite point of view. It's not encyclopedic, and even if it defines an item that overlaps with your API's domain, the application semantics it defines may have nothing to do with how you look at things.

Dublin Core

- **Medium:** HTML, XML, RDF, or plain text
- **Site:** Dublin Core home page
- **Semantics:** published works

The Dublin Core is the original standard for defining application semantics, dating all the way back to 1995. It defines 15 bits of semantics for information about published works: `title`, `creator`, `description`, and so on. These bits of semantics can be used either as semantic descriptors or as link relations.

The Dublin Core Metadata Initiative has also defined a more complete profile, the DCMI Metadata Terms. This profile includes semantic descriptors like `dateCopyrighted`, as well as link relations like `isPartOf` and `replaces`.

Activity Streams

- **Medium:** Atom, JSON
- **Site:** Activity Streams home page
- **Families:** things human beings do online

Activity Streams is a corporate standard for representing our online lives as a sequence of discrete "activities." Each activity has an actor (usually a human being who's using a computer), a verb (something the actor is doing), and an object (the thing to which the actor is doing the verb).

When you watch a video online, that's an activity. You are the actor, the video is the object, and the verb (according to Activity Streams) is the literal string "play." Some activities have a target as well as an object. When I publish a new entry to my blog, I am the actor, the blog entry is the object, the verb is "post," and the target is my blog.

I've put Activity Streams in this section, even though it's a data format, because the data format doesn't define any hypermedia controls. But there are a lot of really useful semantics in here. Activity Streams defines names and semantic descriptors for a lot of the things we interact with online (`Article`, `Event`, `Group`, `Person`). More important, it defines a lot of useful names for verbs (`join`, `rsvp-yes`, `follow`, `cancel`), which make sense as the names of unsafe state transitions.

The Activity Streams standard explains how to represent a sequence of activities as an Atom feed. Use this and Activity Streams will be a real hypermedia format, an extension to Atom.

There's also a standalone JSON-based version of Activity Streams. It has the same problems as GeoJSON: there are no hypermedia controls, and no way to distinguish Activity Streams documents from plain JSON documents.[10] To add hypermedia controls to a JSON Activity Streams document, you'll need to use JSON-LD or Hydra (Chapter 12).

There's a lot of overlap between Activity Streams' semantics and schema.org's microdata items. There are microdata items called Article, Event, Group, and Person. The UserCheckins microdata item is like Activity Streams' "checkin" verb, UserLikes is like "like," and UserPlays is like "play." (For the record, Activity Streams predates schema.org.)

The ALPS Registry

I've set up a registry of ALPS profiles at this page for general reuse. As part of my work to liberate application semantics from their media types, I've created ALPS versions of the schema.org metadata items, several microformats, and the Dublin Core. That's just a start; hopefully by the time you read this I'll have made ALPS profiles that convey the application semantics of other standards as well.

If you want to use an ALPS profile to define your API's application semantics, you can search alps.io to find a profile that works for you, or assemble a new profile out of bits of existing profiles.

If you decide to use an ALPS profile in your API, feel free to reference bits of the profiles in the ALPS Registry. Once you're done, I'd appreciate it if you'd upload the profile to the ALPS Registry (as well as hosting it locally as part of your API). That way other people can find and reuse your application semantics.

10. The Internet-Draft "draft-snell-activity-streams-type" will solve the second problem. It registers the media type `application/stream+json` for Activity Streams documents.