# CHAPTER 3
# Designing Representations

As far as clients are concerned, a *resource* is an abstract entity that is identified by a URI. A *representation*, on the other hand, is concrete and real since that is what you program to and operate upon in clients and servers.

Recall from Recipe 1.1 that HTTP provides an envelope format for representations in requests and responses. Designing a representation involves (a) using that envelope format to include the right headers, and (b) when there is a body for the representation, choosing a media type and designing a format for the body. This chapter presents the following recipes covering various aspects of representation design:

*Recipe 3.1, "How to Use Entity Headers to Annotate Representations"*
: Use this to decide what entity headers to include when sending a representation.

*Recipe 3.2, "How to Interpret Entity Headers"*
: Use this to decide how to interpret entity headers from a representation received.

*Recipe 3.3, "How to Avoid Character Encoding Mismatch"*
: Use this recipe to learn about some precautions about character encoding mismatch.

*Recipe 3.4, "How to Choose a Representation Format and a Media Type"*
: Use this recipe to find the criteria to choose a representation format and a media type.

*Recipe 3.5, "How to Design XML Representations"*
: Use this recipe to decide the essential ingredients for XML-formatted representations.

*Recipe 3.6, "How to Design JSON Representations"*
: Use this recipe to learn how to design JSON-formatted representations.

*Recipe 3.7, "How to Design Representations of Collections"*
: Refer to this recipe to learn about the conventions used to design representations of collections.

*Recipe 3.8, "How to Keep Collections Homogeneous"*
: Use this recipe to check for guidelines on how to keep collections easy to iterate.

*Recipe 3.9, "How to Use Portable Data Formats in Representations"*
: Use this recipe to learn about interoperable ways to format numbers, dates, times, currencies, etc., in representations.

*Recipe 3.10, "When to Use Entity Identifiers"*
: Although URIs are unique identifiers of resources, sometimes using entity identifiers can help improve interoperability. Use this recipe to learn why.

*Recipe 3.11, "How to Encode Binary Data in Representations"*
: Sometimes you may have to deal with binary data. Use this recipe to learn how to use multipart media types to encode binary data in representations.

*Recipe 3.12, "When and How to Serve HTML Representations"*
: When you expect developers or end users to browse certain resources, support HTML format for those resources.

*Recipe 3.13, "How to Return Errors"*
: Errors are also representations, except that they reflect the error state of a resource. Use this recipe to learn how to return error responses.

*Recipe 3.14, "How to Treat Errors in Clients"*
: Use this recipe to learn how to implement clients to process errors.

## 3.1 How to Use Entity Headers to Annotate Representations

A representation is much more than just data serialized in a format. It is a sequence of bytes and metadata that describes those bytes. In HTTP, representation metadata is implemented as name-value pairs using entity headers. These headers are as important as the application data itself. They ensure visibility, discoverability, routing by proxies, caching, optimistic concurrency, and correct operation of HTTP as an application protocol.

### Problem

You want to know what HTTP headers to send in a request to a server or in a response to a client.

### Solution

Use the following headers to annotate representations that contain message bodies:

- `Content-Type`, to describe the type of the representation, including a `charset` parameter or other parameters defined for that media type.
- `Content-Length`, to specify the size in bytes of the body of the representation.
- `Content-Language`, to specify the language if you localized the representation in a language.
- `Content-MD5`, to include an MD5 digest of the body of the representation when the tools/software processing or storing representations may be buggy and need to

provide consistency checks. Note that TCP uses checksums at the transport level for consistency checking.

- `Content-Encoding`, when you encode the body of the representation using `gzip`, `compress`, or `deflate` encoding.
- `Last-Modified`, to specify the last time the server modified the representation or the resource.

## Discussion

HTTP is designed such that the sender can describe the body (also called the *entity body* or *message body*) of the representation using a family of headers known as *entity headers*. With the help of these headers, recipients can make decisions on how to process the body without looking inside the body. These headers also minimize the amount of out-of-band knowledge and guesswork needed to parse the body.

Here is an example of a representation annotated:

```
Content-Type: application/xml;charset=UTF-8
Content-Language: en-US
Content-MD5: bbdc7bbb8ea5a689666e33ac922c0f83
Last-Modified: Sun, 29 Mar 2009 04:51:38 GMT

<user xmlns:atom="http://www.w3.org/2005/Atom">
  <id>user001</id>
  <atom:link rel="self" href="http://example.org/user/user001"/>
  <name>John Doe</name>
  <email>john@example.org</email>
</user>
```

Let's now look at each of the headers.

### Content-Type

This header describes the "type" of a representation and is more generally known as the *media-type* or *MIME type*. Examples include `text/html`, `image/png`, `application/xml`, and `text/plain`. These are all identifiers of the format used to encode the body of the representation. Roughly speaking, a format is the way you encode information into some medium, such as a file, a disk, or the network. XML, JSON, text, CSV, PDF, etc. are formats. A media type identifies the format used and describes the semantics of how to interpret the body of a representation. `application/xml`, `application/json`, `text/plain`, `text/csv`, `application/pdf`, etc., are all media types.

This header informs the receiver of how to parse the data. For instance, if the value of the header is `application/xml` or any value that ends with `+xml`, you can use an XML parser to parse the message. If the value is `application/json`, you can use a JSON parser. When this header is absent, all you are left with is guesswork about the nature of the body.

### Content-Length

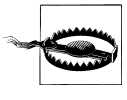
Originally introduced in HTTP 1.0, the purpose of this header is to let the recipient of a message know whether it has read the correct number of bytes from the connection. To send this header, the sender needs to compute the size of the representation before writing the body. HTTP 1.1 supports a more efficient mechanism known as *chunked transfer encoding*. This makes the `Content-Length` header redundant. Here is a representation using chunked encoding:

```
HTTP/1.1 200 OK
Last-Modified: Thu, 02 Apr 2009 02:32:28 GMT
Content-Type: application/xml;charset=UTF-8
Transfer-Encoding: chunked

FF
[some bytes here]

58
[some bytes here]
0
```

Include the `Content-Length` if the client does not support HTTP 1.1.



For `POST` and `PUT` requests, even if you are using `Transfer-Encoding: chunked`, include the `Content-Length` header in requests from client applications. Some proxies reject `POST` and `PUT` requests that contain neither of these headers.

### Content-Language

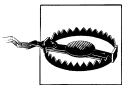
Use this header when the representation is localized for a specific language. The value of this header is a two-letter RFC 5646 language tag, optionally followed by a hyphen (-) and any two-letter country code. Here is an example:

```
# Response
HTTP/1.1 200 OK
Content-Language: kr

<address type="work">
  <street-address>강남구 삼성동 144-19,20 번지 JS 타워</street-address>
  <locality>서울특별시</locality>
  <postal-code>135-090</postal-code>
  <country-name>대한민국</country-name>
  <country-code>KR</country-code>
</address>
```

### Content-MD5

Recipients can use this header to validate the integrity of entity body. The value of this header is an MD5 digest of the body of the representation computed after applying the content encoding (`gzip`, `compress`, etc.) but before applying the transfer encoding (e.g., chunked encoding).



Since this header does not guarantee that the message has not been tampered with, do not use this header as a measure of security. Whoever altered the body can also update the value of this header.

This header can be useful when sending or receiving large representations over potentially unreliable networks. When the sender of a representation includes the `Content-MD5` header, the recipient can verify the integrity of the message before attempting to parse it.

#### Content-Encoding

The presence of this header indicates the type of compression applied to the body of the representation. The value of this header is a string like `gzip`, `compress`, or `deflate`. Here is a `gzip`-encoded representation:

```
Content-Type: application/xml;charset=UTF-8
Content-Language: en-US
Content-MD5: b7c50feb215b112d3335ad0bd3dd88c1
Content-Encoding: gzip
Last-Modified: Sun, 29 Mar 2009 04:51:38 GMT

... gzip encoded bytes ...
```

The recipient of this message needs to decompress this message before parsing the body.

Clients can indicate their preference for `Content-Encoding` using the `Accept-Encoding` header (see Chapter 7 for more details). However, there is no standard way for the client to learn whether a server can process representations compressed in a given encoding.

Unless you know out of band that the target server supports a particular encoding method, avoid using this header in HTTP requests.

#### Last-Modified

This header applies for responses only. This value of this header is the timestamp of the last time the server modified the representation of the resource. We will discuss this header in Chapter 9.

## 3.2 How to Interpret Entity Headers

When a server or client receives a representation, correctly interpreting entity headers before processing a request is vital. This recipe discusses how to interpret a representation from the headers included.

### Problem

You want to know how to interpret the entity headers included in a representation, and how to process the representation using those headers.

### Solution

*Content-Type*
: When you receive a representation with no `Content-Type`, avoid guessing the type of the representation. When a client sends a request without this header, return error code `400` (`Bad Request`). When you receive a response without this header from a server, treat it as a bad response.

*Content-Length*
: Do not check for the presence of the `Content-Length` header in a representation you receive without first confirming the absence of `Transfer-Encoding: chunked`.

*Content-Encoding*
: Let your network library deal with uncompressing compressed representations.

*Content-Language*
: Read and store the value of this header, if present, to record the language used.

### Discussion

In most cases, client applications need only deal with checking the `Content-Type` header and character encoding to determine how to parse the body of a representation. Client-side HTTP libraries must be able to deal with `Content-Encoding` transparently.

Some software applications assume that the `Content-Length` header must always be present and reject representations that do not contain this header. This is an incorrect assumption. If you must determine the message length before processing a request or a response in your code, follow the procedure outlined in Section 4.4 of RFC 2616.

Make sure to process representations in responses based on the values of the `Content-Type`, `Content-Language`, and `Content-Encoding` headers. For instance, just because the client sent an `Accept: application/json` header or because the URI for the resource ends with `.json`, don't assume the response will be JSON formatted. See Recipe 7.1 for how to inform the server of what types of representations the client can process.

## 3.3 How to Avoid Character Encoding Mismatch

Character encoding mismatch between the sender and receiver of a representation usually results in data corruption and often in parse errors.

## Problem

You want to know how to ensure that the characters in your representations are interpreted correctly by the recipients.

## Solution

When sending a representation, if the media type allows a `charset` parameter, include that parameter with a value of the character encoding used to convert characters into bytes.

When you receive a representation with a media type that supports the `charset` parameter, use the specified encoding when constructing a character stream from bytes in the body of the representation. If you ignore the sender-supplied `charset` value and use some other value, your applications may misinterpret the characters.

If you receive an XML, JSON, or HTML representation with a missing `charset` parameter, let your XML, JSON, or HTML parsers interpret the character set by inspecting the first several bytes as per algorithms outlined in specifications of those formats.

## Discussion

Text and XML media types such as `application/xml`, `text/html`, `application/atom+xml`, and `text/csv` let you specify the character encoding used to convert characters into bytes in the entity body via a `charset` parameter of the `Content-Type` header. Here is an example:

```
Content-Type: application/xml;charset=UTF-8
```

The JSON media type `application/json` does not specify a `charset` parameter but uses `UTF-8` as the default encoding. RFC 4627 specifies ways to determine the character encoding of JSON-formatted data.

Errors due to character encoding mismatch can be hard to detect. For instance, when a sender uses `UTF-8` encoding to encode some text into bytes and the recipient uses `Windows-1252` encoding to decode those bytes into text, you will not detect any issues as long as the characters the sender used have the same code values in both the encodings. For instance, a phrase such as "`Hello World`" will appear the same on both sides, but a phrase such as "`2 €s for an espresso?`" will appear as "`2 ?Ks for an espresso?`" because of differences between these encodings.

Such mismatch is described by the term *Mojibake*. See *http://en.wikipedia.org/wiki/Mojibake* for more examples.

Another common way to introduce a character encoding mismatch in XML representations is to report one encoding in the `Content-Type` header and report another in the body as in the following example:

```
Content-Type: application/xml; charset=UTF-8 ❶

<?xml version="1.0" encoding="ISO-8859-1"?> ❷
<user> ... </user>
```

❶ `UTF-8` declared in the `Content-Type` header

❷ `ISO-8859-1` declared in the prolog of the XML document

In this case, if you ignore supplying the encoding from the `charset` parameter (`UTF-8`) to the XML parser, the parser will attempt to determine the character encoding from the prolog and will find it as `ISO-8859-1`. This will cause the recipient to misinterpret the characters in the body.

Also avoid using the `text/xml` media type for XML-formatted representations. The default charset for `text/xml` is `us-ascii`, whereas `application/xml` uses `UTF-8`.

## 3.4 How to Choose a Representation Format and a Media Type

This may be one of the first questions to come to mind when designing a RESTful web service. However, no single format may be right for all kinds of resources and representations. Picking up a format like JSON or XML for all representations may reduce the flexibility that HTTP has to offer.

### Problem

You want to know how to choose a format and a media type for representations.

### Solution

Keep the choice of media types and formats flexible to allow for varying application use cases and client needs for each resource.

Determine whether there is a standard format and media type that matches your use cases. The best place to start your search is the Internet Assigned Numbers Authority (IANA, *http://www.iana.org/assignments/media-types/*) media type registry.

If there is no standard media type and format, use extensible formats such as XML (`application/xml`), Atom Syndication Format (`application/atom+xml`), or JSON (`application/json`).

Use image formats like `image/png` or rich document formats like `application/vnd.ms-excel` or `application/pdf` to provide alternative representations of data. When using such formats, consider adding a `Content-Disposition` header, as in

`Content-Disposition: attachment; filename=<status.xls>` to give a hint of the filename that the client could use to save the representation to the filesystem.

Prefer to use well-known media types for representations. If you are designing a new media type, register the format and media type with IANA by following the procedure outlined in RFC 4288.

## Discussion

HTTP's message format is designed to allow different media types and formats for requests and responses. Some resources may require XML-formatted representations, others may require HTML representations, while still others may require PDF-formatted representations. Similarly, some resources can process `application/x-www-form-urlencoded` but return XML-formatted representations in response. Leaving room for such flexibility is a vital part of designing representations. For instance, a system managing customer accounts may need to provide a variety of media types and formats.

- An XML-formatted representation for each customer account
- An Atom feed of all new customers
- Customer trends presented as a spreadsheet
- HTML pages for summary of each customer

When it comes to format and media type selection, the rule of thumb is to let the use cases and the types of clients dictate the choice. For this reason, it is important not to pick up a development framework that rigidly enforces one or two formats for all resources with no flexibility to use other formats.

### Using standard or well-known media types

When selecting a format and a media type for representations, first check whether there is a standard or well-known format and media type that matches your use cases. The IANA media type registry lists media types by primary types such as `text` and `application` and subtypes such as `plain`, `html`, and `xml` and provides additional references to the media type and the underlying format. For example, at *http://www.iana.org/assignments/media-types/application/*, you will find that RFC 4627 defines the media type `application/json`. If you decide to use JSON as a format for your representations, that is the document to consult to learn the semantics of this format. Table 3-1 lists some commonly used standard or well-known media types.

*Table 3-1. Well-known/standard media types*

| Media types | Format | Reference |
|---|---|---|
| application/xml | Generic XML format | RFC 3023 |
| application/*+xml | Special-purpose media types using the XML format | RFC 3023 |
| application/atom+xml | An XML format for Atom documents | RFC 4287 and RFC 5023 |

| Media types | Format | Reference |
|---|---|---|
| application/json | Generic JSON format | RFC 4627 |
| application/javascript | JavaScript, for processing by JavaScript-capable clients | RFC 4329 |
| application/x-www-form-urlencoded | Query string format | HTML 4.01 |
| application/pdf | PDF | RFC 3778 |
| text/html | Various versions of HTML | RFC 2854 |
| text/csv | Comma-separated values, a generic format | RFC 4180 |

In this table, the first column specifies the media type, whereas the second one specifies the format used by the media type. *The generic formats in this table have no application-specific semantics*. For example, an XML-formatted representation for a customer account resource will have widely different semantics than, say, an XML-formatted representation for a purchase order resource. In this example, it is up to the server to define the semantics of various XML elements in these representations.

```
# A customer representation
Content-Type: application/xml;charset=UTF-8

<customer>
  <id>urn:example:customer:cust001</id>
  ...
</customer>

# A purchase customer representation
Content-Type: application/xml;charset=UTF-8

<po>
  <id>urn:example:po:po001</id>
  ...
</po>
```

On the other hand, specialized formats such as Atom, PNG, HTML, and PDF have concrete semantics specified by the respective RFCs or other documents listed in Table 3-1. Take, for example, the following HTML representation of a customer:

```
# A customer representation
Content-Type: text/html;charset=UTF-8

<html>
  <head>
    <title>Customer Xyz</title>
  </head>
  <body>
    ...
  </body>
</html>
```

The HTML specifications describe the semantics of this representation. If you decide to use a generic format such as XML or JSON, you should document the semantics of the representations in as much detail as possible.

#### Introducing new formats and media types

You can design completely new textual or binary formats with application-specific rules for encoding and decoding data, and you can assign new media types for those formats. For instance, you can assign the media type `application/vnd.example.customer+xml` for the XML format used for customer account resource. Here `vnd` stands for "vendor," implying that this is a vendor/implementation-specific media type:

```
# A customer representation
Content-Type: application/vnd.example.customer+xml;charset=UTF-8

<customer>
  <id>urn:example:customer:cust001</id>
  ...
</customer>
```

In this case, by looking at the `Content-Type` header and without parsing the XML, any software that is aware of this media type can recognize that this is a customer account representation. The following two things may motivate the introduction of such new media types:

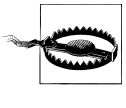
*New formats*
: In some cases, your application data may be specialized and significantly differs from any existing related media types. Examples include new audio, video, or document formats or binary formats for encoding data.

*Visibility*
: As shown in the previous example, application-specific media types promote visibility as long as such media types are widely supported.

If you choose to create new media types of your own, consider the following guidelines:

- If the media type is XML based, use a subtype that ends with `+xml`.
- If the media type is for private use, use the subtype starting with `vnd.`. For example, you can use a media type such as `application/vnd.example.org.user+xml`. This is another convention used by some application-specific media types.
- If the media type is for public use, register your media type with IANA as per RFC 4288.

Note that new media types that are not widely recognized may reduce interoperability with clients as well as tools such as proxies, log file analyzers, monitoring software, etc.



Avoid introducing new application-specific media types unless they are expected to be broadly used. Proliferation of new application-specific media types may impede interoperability.

Although custom media types improve protocol-level visibility, existing protocol-level tools for monitoring, filtering, or routing HTTP traffic pay little or no attention to media types. Hence, using custom media types only for the sake of protocol-level visibility is not necessary.

## 3.5 How to Design XML Representations

For representations that are application specific, such as a customer profile or a purchase order, it is natural to include application data in representations. In addition, in order to make representations in your web service consistent with each other and to improve the usability of those representations, it is essential that you include certain additional details in each representation.

### Problem

You want to know what data to include in XML-formatted representations.

### Solution

In each representation, include a self link (i.e., a link with the link relation type `self`) to the resource (see Chapter 5), and include identifiers for each of the application domain entities that makes up a resource (Recipe 3.10).

If part of the representation contains natural-language text, add `xml:lang` attributes indicating the language that the contents of that element are localized in.

### Discussion

Including common elements such as identifiers and links in all representations makes it easier for clients and servers to process requests and generate responses. For instance, the self link can help clients know the URI for the representation, and clients can use that as an identifier for the resource.

The self link serves the same purpose as the request URI when the response contains the representation of the resource at that URI, or as the `Content-Location` header when the representation in the response does not correspond to the resource at the request URI. For instance, in the first request shown here, the request URI corresponds to the location of the resource for the response in the representation. In the second request, the `Content-Location` provides the URI of the resource:

```
# Request
GET /user/smith/address/0 HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
```

```
<address>
  <id>urn:example:user:smith:address:0</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/0"/>
  <street>1, Olympia Dr</street>
  <city>Some City</city>
</address>


# Second request to create a resource
POST /user/smith HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<address>
  <street>1, Main Street</stret>
  <city>Some City</city>
</address>

# Response
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/1
Content-Location: http://www.example.org/user/smith/address/1
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/1"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

Including self links in the body of the representation may be useful when the code used for processing the body does not have access to the request URI or the response headers.

For representations that contain data localized in more than one language, the `Content-Language` header is not sufficient. In such cases, include language tags directly to the body of the representation. Here is an example, adapted from the XML 1.0 specification:

```
# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Content-Language: en

<content>
  <text>The quick brown fox jumps over the lazy dog.</text> ❶
  <text xml:lang="en-GB">What colour is it?</text> ❷
  <text xml:lang="en-US">What color is it?</text> ❸
  <text xml:lang="de"> ❹
    <p>Habe nun, ach! Philosophie,</p>
    <p>Juristerei, und Medizin</p>
```

```
      <p>und leider auch Theologie</p>
      <p>durchaus studiert mit heißem Bemüh'n.</p>
    </text>
  </content>
```

❶ Text in the default language for the representation, as specified by the `Content-Language` header

❷ Text in the `en-GB` language

❸ Text in the `en-US` language

❹ Text in all the child elements in the `de` language

## 3.6 How to Design JSON Representations

JSON is a JavaScript-based data format. Like XML, it is a general-purpose, human-readable, and extensible format. In languages like JavaScript and PHP, parsing JSON structures is easier than parsing XML. Most web services that are consumed by browser-based clients often prefer JSON over representation formats.

### Problem

You want to know what data to include in JSON-formatted representations.

### Solution

In each representation, include a self link to the resource (see Recipe 5.2), and include identifiers for each of the application domain entities that make up resource (Recipe 3.10).

If an object in the representation is localized, add a property to indicate the language its contents are localized in.

### Discussion

The approach presented in this recipe is similar to that of XML (Recipe 3.5). Here is an example of a representation of a person resource:

```
{
  "name" : "John",
  "id" : "urn:example:user:1234",
  "link" : {
     "rel : "self",
     "href" : "http://www.example.org/person/john"
  },
  "address" : {
    "id" : "urn:example:address:4567",
    "link" : {
       "rel : "self",
       "href" : "http://www.example.org/person/john/address"
    }
```

```
    ...
  }
}
```

When the `Content-Language` header does not sufficiently describe the locale of the representation, add a property to express the language, as in the following example:

```
{
  "content" : {
    "text" : [{
         "value" : "The quick brown fox jumps over the lazy dog."
       },
       {
         "lang" : "en-GB",
         "value" : "What colour is it"
       },
       {
         "lang" : "en-US",
         "value" : "What color is it"
       }
     ]
  }
}
```

## 3.7 How to Design Representations of Collections

Clients use collections to iterate through its members. Since some collections contain a large number of member resources, clients need a way to paginate/scroll through the collection.

### Problem

You want to know what to include in representations of collection resources.

### Solution

Include the following in each collection representation:

- A self link to the collection resource
- If the collection is paginated and has a next page, a link to the next page
- If the collection is paginated and has a previous page, a link to the previous page
- An indicator of the size of the collection

### Discussion

A collection resource is like any other resource except that, in some case, it contains a large number of members. When a server returns only a subset of members in the representation of a collection, the server should also provide links to allow the client to paginate through all the members. Here is a collection resource containing several articles:

```
# Request
GET /articles?contains=cycling&start=10 HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Content-Language: en

<articles total="1921" xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self"
    href="http://www.example.org/articles?contains=cycling&start=10"/> ❶
  <atom:link rel="prev"
    href="http://www.example.org/books?contains=cycling"/> ❷
  <atom:link rel="next"
    href="http://www.example.org/books?contains=cycling&start=20"/> ❸
  <article>
    <atom:link rel="self"
      href="http://www.nytimes.com/2009/07/15/sports/cycling/15tour.html"/>
    <title>For Italian, Yellow Jersey Is Fun While It Lasts</title>
    <body>...</body>
  </article>
  <article>
    <atom:link rel="alternate"
      href="http://www.nytimes.com/2009/07/27/sports/cycling/27tour.html"/>
    <title>Contador Wins, but Armstrong Has Other Victory</title>
    <body>...</body>
  </article>
  ...
</articles>
```

❶ A link to the collection itself

❷ A link to the previous page

❸ A link to the next page

This representation is the result of searching a large collection of news articles. This representation has three links—a link with the `self` relation type to get the representation itself, a link with the `prev` relation type to get the previous 10 articles, and another link with the `next` relation type to get the next 10 articles. Clients can use these links to navigate through the entire collection.

The `total` attribute gives the client an indication of the number of members in the collection.

Although the size of the collection is useful for building user interfaces, avoid computing the exact size of the collection. It may be expensive to compute, volatile, or even confidential for your web service. Providing a hint is usually good enough.

At the HTTP level, each page is a different resource. This is because each page of results in this example has a different URI such as `http://www.example.org/books?contains=cycling` and `http://www.example.org/books?contains=cycling&start=10`.

## 3.8 How to Keep Collections Homogeneous

Depending on use cases, you can group resources into collections by using similarities. However, no matter what criteria you choose for any collection, it is important to keep the representation homogeneous so that it is easy to use by clients.

### Problem

You want to know how to design a representation format for a collection whose members don't completely look alike.

### Solution

Design the representation of the collection such that members in a collection are structurally and syntactically similar.

### Discussion

When designing a representation format for the collection, include only the homogeneous aspects of its member resources. For instance, if your collection of products can contain cars, boats, and motorcycles, include just the product-specific details of those resources in the product collection. Note that collections are meant for grouping resources that are similar in some sense, and when you include resource-specific information that is not common across other resources within the same collection, it usually is a result of poor abstraction. Here is an example of such a poor abstraction:

```
<!-- Avoid this -->
<products xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/catalog/products"/>
  <!-- The first member is an automobile. -->
  <automobile>
    <id>9001</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/9001"/>
    <make>Smart</make>
    <model>Fortwo Convertible</model>
    <year>2009</year>
    <class classid="small">Small Car</class>
    <mpg>
      <city>33</city>
      <highway>41</highway>
    </mpg>
    <drivetrain>2WD</drivetrain>
    <list-price currency="USD">19495</list-price>
  </automobile>
  <!-- The second member is a boat! -->
  <sailboat>
```

```
    <id>10101</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/10101"/>
    <make>Jeanneau</make>
    <model>Sunfast 3200</model>
    <year>2008</year>
    <length unit="ft">32</length>
    <hull-type>fiberglass</hull-type>
    <number-of-engines>1</number-of-engines>
    <list-price currency="USD">95995</list-price>
  </sailboat>
</products>
```

In this example, although the automobile and sailboat share common properties, they have properties that are specific to each product. For a client application iterating over such a collection, those specific properties may not make sense, and clients may not be able to cope with such a representation. Consider avoiding such representations, and keep the representation of collections homogeneous, as in the following:

```
<products xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/catalog/products"/>
  <product type="automobile">
    <id>9001</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/9001"/>
    <make>Smart</make>
    <model>Fortwo Convertible</model>
    <year>2009</year>
    <list-price currency="USD">19495</list-price>
  </product>
  <product type="sailboat">
    <id>10101</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/10101"/>
    <make>Jeanneau</make>
    <model>Sunfast 3200</model>
    <year>2008</year>
    <list-price currency="USD">95995</list-price>
  </product>
</products>
```

This homogeneous form is more convenient for the client than the previous example.

## 3.9 How to Use Portable Data Formats in Representations

There are a number of ways to encode dates, times, countries, numbers, and time zones in representations. For instance, you can format date-time values as using the Unix `date` format, Unix epoch time, or plain MM-DD-YYYY and DD-MM-YYYY formats. Most date-time formats require clock synchronization between clients and servers or depend on local time. These formats cause interoperability problems because of differences in clocks, time zones, or even daylight saving time. Similarly, currency and number formats vary from country to country, and representations designed for audiences in one country may not interoperate with clients or servers in another country unless you use portable data formats.

## Problem

You want to know the appropriate formats to choose for dates, times, numbers, currencies, etc.

## Solution

Except when the text is meant for presentation to end users, avoid using language-, region-, or country-specific formats or format identifiers. Instead, use the following portable formats:

- Use decimal, float, and double datatypes defined in the W3C XML Schema for formatting numbers including currency.
- Use ISO 3166 codes for countries and dependent territories.
- Use ISO 4217 alphabetic or numeric codes for denoting currency.
- Use RFC 3339 for dates, times, and date-time values used in representations.
- Use BCP 47 language tags for representing the language of text.
- Use time zone identifiers from the Olson Time Zone Database to convey time zones.

## Discussion

Choosing portable formats for data eliminates interoperability errors. See examples below for some commonly used formats. Note that your application domain may involve additional types of data not included here. Look for industry- or domain-specific standards before inventing your own.

### Numbers

The formats specified by the XML Schema for numbers are language and country independent and hence are portable:

```
123.456   +1234.456   -1234.456   -.456, 123.
```

However, formats like the following are not portable:

```
1,234,567   12,34,567   1,234
```

### Countries and territories

ISO 3166-1, the first part of ISO 3166, specifies two-letter country codes such as `US` for the United States, `Dk` for Denmark, `IN` for India, etc.

ISO 3166-2, the second part of ISO 3166, specifies codes for subdivisions of countries such as states and provinces. Examples include `US-WA`, `US-CO`, `CA-BC`, `IN-AP`, etc.

### Currencies

ISO 4217 specifies three-letter currency codes for names of currencies. The first two letters of these codes represent ISO 3166-1 two-letter country codes, and the third letter is usually the initial for the currency. Examples include `USD` for the U.S. dollar, `CAD` for the Canadian dollar, and `DKK` for the Danish krone. These codes represent revaluation and changes in currencies. Using these codes with currency values removes ambiguity with currency names such as "dollar" or symbols such as $.

### Dates and times

RFC 3339 is a profile of ISO 8601, which is a standard for representing dates and times using the Gregorian calendar. RFC 3339–formatted dates, times, and date-time values have the following characteristics:

- You can compare two values by sorting them as strings.
- This format is human readable.
- Dates can use either Coordinated Universal Time (UTC) or an offset from the UTC, thus avoiding issues related to time zones and daylight saving time.

Here are some examples of properly formatted date, time, and date-time values:

```
2009-09-18Z
23:05:08Z
2009-09-18T23:05:08Z
2009-09-18T23:05:08-08:00
```

The `date`, `time`, and `dateTime` datatypes in the W3C XML Schema follow RFC 3339, and you can use libraries that support these datatypes to read and parse these values.

### Language tags

BCP stands for "best current practice." BCP 47 currently refers to RFC 5646 and RFC 5645 that define values of language tags such as the HTML `lang` attribute and XML `xml:lang` attribute. Examples include `en` for English, `en-CA` for Canadian English, and `ja-JP` for Japanese as used in Japan.

### Time zone identifiers

The Olso Time Zone Database provides a uniform convention for time zone names and contains data about time zones. This database accounts for time zones, seasonal changes such as daylight saving time, and even historical time zone changes. Most programming languages support time zone classes/utilities that support this database. Examples include Java's `java.util.Timezone`, Ruby's `TZInfo`, Python's `tzinfo`, and C#'s `System.TimeZoneInfo`.

## 3.10 When to Use Entity Identifiers

For RESTful web services, URIs are the unique identifiers for resources. However, application code usually has to deal with identifiers of domain entities. When a client or a server is part of a larger heterogeneous set of applications, information from resources may cross several system boundaries, and entity identifiers can be used to cross-reference or transform data.

### Problem

You want to know when to include entity identifiers in representations along with resource URIs.

### Solution

For each of the application domain entities included in the representation of a resource, include identifiers formatted as URNs.

### Discussion

Although URIs uniquely identify resources, entity identifiers come in handy for the following:

- When your clients and servers are part of a larger environment containing applications using RPC, SOAP, asynchronous messaging, stored procedures, and even third-party applications, entity identifiers may be the only common denominator across all those systems to provide the identity of data uniformly.
- Clients and servers can maintain their own stored copies of entities included in a resource without having to decode from resource URIs or having to use URIs as database keys. Although not ideal, URIs may change. Clients can use these identifiers to cross-reference various entities referred to from different representations.
- When not all entities in your application domain are mapped to resources, entity identifiers can help provide uniqueness for data contained in representations.

Even if you mapped all the entities in your application to resources with unique URIs, including entity identifiers in representations will future-proof your application when it needs to integrate with non-HTTP web services. To maintain the uniqueness of identifiers, consider formatting identifiers as URNs.

Here is an example, where the database identifier of the user resource is `1234` and that of the user's address is `4567`:

```
<person xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://example.org/person/john"/>
  <id>urn:example:user:1234</id>
  <name>John Doe</name>
  <address>
    <id>urn:example:address:4567</id>
```

```
    <street>1 Main Street</street>
    <city>Seattle</city>
    <state>WA</state>
  </address>
</person>
```

## 3.11 How to Encode Binary Data in Representations

Not every representation can completely rely on textual formats such as XML and JSON. Some representations may need to contain binary data within textual representations. Examples include a video preview of a movie in a movie catalog or some image cover art of a representation of an audio sample in a music store.

### Problem

You want to know how to encode binary data in representations that also contain textual data.

### Solution

Use multipart media types such as `multipart/mixed`, `multipart/related`, or `multipart/alternative`. Avoid encoding binary data within textual formats using Base64 encoding.

### Discussion

Multipart messages give you the ability to combine dissimilarly formatted data into one single HTTP message. A multipart message is a message containing several message parts each separated by a boundary. Each part can contain a message of a different media type. Here is an example:

```
Content-type: multipart/mixed; boundary="abcd"

--abcd
Content-Type: application/xml;charset=UTF-8

<movie> ... </movie>
--abcd
Content-type: video/mpeg

... image here ...

--abcd--
```

This multipart message has two parts, one containing an XML document and the other containing a video. Consider one of the multipart media types listed in Table 3-2 for such use cases.

*Table 3-2. Using multipart media types*

| Media type | Usage |
|---|---|
| `multipart/form-data` | To encode name-value pairs of data mixed with parts containing data of arbitrary media types. The usage is the same as you would use to upload files using HTML forms. |
| `multipart/mixed` | To bundle several parts of arbitrary media types. In the previous example, the multipart message combined the metadata of a movie represented as `application/xml` and the video as `video/mpeg` into a single HTTP message. |
| `multipart/alternative` | Use this when sending alternative representations of the same resource using different media types. The best example for this media type is sending email as plain text (media type `text/plain`) and HTML (media type `text/html`). |
| `multipart/related` | Use this when the parts are interrelated and you need to process the parts together. The first part is the root part and can refer to various other parts via a `Content-ID` header. |

Creating and parsing multipart messages in some programming languages may be cumbersome and complex. As an alternative, instead of including binary data in representations, provide a link to fetch the binary data as a separate resource. For instance, in the previous example, you can provide a link to the video.

# 3.12 When and How to Serve HTML Representations

HTML is a popular hypermedia format, and with browsers as universal clients, users can interact with HTML representations without any application-specific logic implemented in browsers. Moreover, you can use JavaScript and HTML parsers to extract or infer data from HTML. This recipe discusses the pros and cons and when HTML may be appropriate.

## Problem

You want to know if you must design HTML representations along with XML or JSON-formatted representations, and if so, how.

## Solution

For resources that are expected to be consumed by end users, provide HTML representations. Avoid designing HTML representations for machine clients. To enable web crawlers and such software, use microformats or RDFa to annotate data within the markup.

## Discussion

HTML is widely understood and supported by client software such as browsers, HTML parsers, authoring tools, and generating tools. It is also self-describing, enabling users to use any HTML-compliant client to interact with servers. This makes it a suitable

format for human consumption. For instance, consider the following XML-formatted representation of a resource:

```
<person xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://example.org/person/john"/>
  <id>urn:example:user:1234</id>
  <name>John</name>
  <address>
     <atom:link rel="self" href="http://example.org/person/john"/>
     <id>usr:example:address:4567</id>
     <street>1 Main Street</street>
     <city>Seattle</city>
     <state>WA</state>
  </address>
</person>
```

You can design the following equivalent HTML representation (without any CSS styles, for the sake of simplicity) for the same resource:

```
<html>
  <head>
    <title>John</title>
    <link rel="self" href="http://example.org/person/john"/>
  </head>
  <body>
    <h1>John</h1>
    <div>
      <div>1 Main Street</div>
      <div>Seattle</div>
      <div>WA</div>
    </div>
  </body>
</html>
```

When offering some or all of your representations as HTML documents, consider annotating HTML with microformats or RDFa. Doing so allows web crawlers and such software to extract information from your HTML documents without depending on the structure of your HTML documents. Here is an example of the previous HTML representation annotated with the `hcard` microformat (*http://microformats.org/wiki/hcard*):

```
<html>
  <head>
    <title>John</title>
  </head>
  <body>
    <h1 class="fn">John</h1>
    <div class="vcard">
      <div class="adr">
        <div class="street-address">1 Main Street</div>
        <div class="locality">Seattle</div>
        <div><abbr class="region" title="Washington">WA</abbr></div>
      </div>
    </div>
```

```
  </body>
</html>
```

Microformats use HTML `class` attributes to annotate various HTML elements so that HTML-aware clients can interpret the semantics of those elements. The `hcard` microformat is a mapping of the `vcard` format (RFC 2426) to HTML. The `vcard` format is an interoperable standard for representing addresses. The `hcard` microformat specifies several CSS class names. The previous example uses the class name `fn` for the name, `adr` for the address, `street-address` for street names, `locality` for location names, and `region` for regions such as states.

Any microformat-capable HTML parser can interpret the address from this HTML document. Adding this format need not affect the rendering of the document in browsers since microformats use the `class` attribute to extend HTML.

You can similarly use RDFa:

```
<html>
  <head>
    <title>John</title>
  </head>
  <body>
    <div xmlns:v="http://www.w3.org/2001/vcard-rdf/3.0#"
         about="http://example.org/person/john">
      <h1 property="v:FN" href="http://example.org/person/john">John</h1>
      <div role="v:ADR">
        <div property="v:Street">1 Main Street</div>
        <div property="v:Locality">Seattle</div>
        <div><abbr property="v:Region" title="Washington">WA</abbr></div>
      </div>
    </div>
  </body>
</html>
```

The only difference is that it uses RDFa and the `vcard` format to annotate HTML elements. Some search engines use these annotations to decipher the semantics of information from HTML documents.

Note that RDFa is specified only for XHTML 1.1. However, all currently deployed browsers do support RDFa for HTML documents.

## 3.13 How to Return Errors

HTTP is based on the exchange of representations, and that applies to errors as well. When a server encounters an error, either because of problems with the request that a client submitted or because of problems within the server, always return a representation that reflects the state of the error condition. This includes the response status code, response headers, and a body containing the description of the error.

## Problem

You want to know how to return errors to clients.

## Solution

For errors due to client inputs, return a representation with a `4xx` status code. For errors due to server implementation or its current state, return a representation with a `5xx` status code. In both cases, include a `Date` header with a value indicating the date-time at which the error occurred.

Unless the request method is `HEAD`, include a body in the representation formatted and localized using content negotiation (see Chapter 7) or in human-readable HTML or plain text.

If information to correct or debug the error is available as a separate human-readable document, include a link to that document via a `Link` header (see Recipe 5.3) or a link in the body.

If you are logging errors on the server side for later tracking or analysis, provide an identifier or a link that can be used to refer to that error. For instance, clients can report the error code to the server's team while reporting problems.

Keep the response body descriptive, but exclude details such as stack traces, errors from database connection failures, etc. If appropriate, describe any actions that the client can take to correct the error or to help the server debug and fix the errors.

## Discussion

HTTP 1.1 defines two classes of error codes, one in the range of `400` to `417` and the other in the range of `500` to `505`. One common mistake that some web services make is to return a status code that reflects success (status codes from `200` to `206` and from `300` to `307`) but include a message body that describes an error condition.

```
# Avoid returning success code with an error in the body.
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<error>
  <message>Account limit exceeded.</message>
</error>
```

Doing this prevents HTTP-aware software from detecting errors. For example, a cache will store it as a successful response and serve it to subsequent clients even when clients may be able to make a successful request.

#### Errors due to client inputs: 4xx

The following list shows error codes you are likely to generate in your server-side application code and not codes that will be automatically by generated by your web/application server:

`400` (`Bad Request`)
: You can return this error when your server cannot decipher client requests becasue of syntactical errors.

  HTTP 1.1 defines only one condition under which you can return this error. That is when the request does not include a `Host` header.

`401` (`Unauthorized`)
: Return this when the client is not authorized to access the resource but may be able to gain access after authentication. If your server will not let the client access the resource even after authentication, then return `403` (`Forbidden`) instead.

  When returning this error code, include a `WWW-Authenticate` header field with the authentication method to use. Commonly used methods are `Basic` and `Digest`, as discussed in Chapter 12.

`403` (`Forbidden`)
: Use this when your server will not let the client gain access to the resource and authentication will not help.

  For instance, you can return this when the user is already authenticated but is not allowed to request a resource.

`404` (`Not Found`)
: Return this when the resource is not found. If possible, specify a reason in the message body.

`405` (`Not Allowed`)
: Return this when an HTTP method is not allowed for this resource.

  Return an `Allow` header with methods that are valid for this resource (see Recipe 14.2).

`406` (`Not Acceptable`)
: See Recipe 7.7.

`409` (`Conflict`)
: Return this when the request conflicts with the current state of the resource. Include a body explaining the reason.

`410` (`Gone`)
: Return this when the resource used to exist, but it does not anymore.

  You may not be able to return this code unless you have some bookkeeping data about deleted resources. If you do not keep track of deleted resources on the server side, return a `404` (`Not Found`) instead.

`412` (`Precondition Failed`)
See Recipe 10.4.

`413` (`Request Entity Too Large`)
Return this when the body of a `POST` of `PUT` request is too large.

If possible, specify what is allowed in the body, and provide alternatives.

`415` (`Unsupported Media Type`)
Return this error when a client sends the message body in a format that the server does not understand.

#### Errors due to server errors: 5xx

The following list shows error codes that you may generate when the request fails because of some error on the server:

`500` (`Internal Server Error`)
This is the best code to return when your code on the server side failed due to some implementation bug.

`503` (`Service Unavailable`)
Return this when the server cannot fulfill the request either for some specific interval or for an undetermined amount of time.

Two common conditions that prompt this error are failures with backend servers (such as a database connection failure) or when the client exceeded some rate limit set by the server.

If possible, include a `Retry-After` response header with either a date or a number of seconds as a hint.

HTTP status codes are normative, but the status messages are not. Those are the messages that HTTP 1.1 uses. Servers are free to use application-specific error message strings.

#### Message body for errors

Include a body in the error response for all errors except when the HTTP method is `HEAD`. In the body, include some or all of the following:

- A brief message describing the error condition
- A longer description with information on how to fix the error condition, if applicable
- An identifier for the error
- A link to learn more about the error condition, with tips on how to resolve it

Here is an example. This is an error that occurred when the client sent a request for an account transfer:

```
# Response
HTTP/1.1 409 Conflict
Content-Type: application/xml;charset=UTF-8
Content-Language: en
Date: Wed, 14 Oct 2009 10:16:54 GMT
Link: <http://www.example.org/errors/limits.html>;rel="help"

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Account limit exceeded. We cannot complete the transfer due to
  insufficient funds in your accounts</message>
  <error-id>321-553-495</error-id>
  <account-from>urn:example:account:1234</account-from>
  <account-to>urn:example:account:5678</account-to>
  <atom:link href="http://example.org/account/1234"
             rel="http://example.org/rels/transfer/from"/>
  <atom:link href="http://example.org/account/5678"
             rel="http://example.org/rels/transfer/to"/>
</error>
```

When generating the message body, consider following the recipes discussed Chapter 7.

## 3.14 How to Treat Errors in Clients

When implementing a client, there are two kinds of errors that the client needs to deal with. The first is network-level failures. The second is HTTP errors returned by servers. Programming libraries deal with the former class of errors and surface them via programming language-specific exception handling. The latter class is application specific and requires explicit coding.

### Problem

You want to know how to interpret errors returned by the server.

### Solution

See the following list for appropriate action for each error code:

`400` (`Bad Request`)
: Look into the body of the error representation on hints for the root cause of the problem.

`401` (`Unauthorized`)
: If the client is user-facing, prompt the user to supply credentials. In other cases, obtain the necessary security credentials. Retry the request with an `Authorization` header containing the credentials.

`403` (`Forbidden`)
: This error means that the client is forbidden from accessing the resource with the request method. Do not repeat the request that caused this error.

`404` (`Not Found`)
: The resource is gone. If you stored data about the resource on the client side, clean up the data or mark it as deleted.

`405` (`Not Allowed`)
: Look for the `Allow` header for the methods that are valid for this resource, and make necessary code changes to limit access to only those methods.

`406` (`Not Acceptable`)
: See Recipe 7.7.

`409` (`Conflict`)
: Look for the conflicts listed in the body of the representation of `PUT`.

`410` (`Gone`)
: Treat this the same as `404` (`Not Found`).

`412` (`Precondition Failed`)
: See Recipe 10.4.

`413` (`Request Entity Too Large`)
: Look for hints on valid size in the body of the error.

`415` (`Unsupported Media Type`)
: See the body of the representation to learn the supported media types for the request.

`500` (`Internal Server Error`)
: Log this error, and then notify the server developers.

`503` (`Service Unavailable`)
: If the response has a `Retry-After` header, avoid retrying until that period of time. This error may be serverwide, and hence, you may need to implement appropriate back-off logic in your clients to avoid sending requests to the server for some period of time.

## Discussion

Explicitly handling various error codes makes clients robust. In particular, watch out for HTTP client libraries that translate both network-level failures and HTTP errors into exception or error classes. These classes of errors need different treatments.

HTTP status codes are extensible, and servers can introduce new status codes. If a client does not understand an `Xmn` status code where `X` is `2`, `3`, `4`, or `5`, then it should treat it as an `X00` code. For example, if a server returns `599` and if the client does not understand what it is, treat it as `500`. The same goes for a status code like `245`.

Do not treat HTTP errors as I/O or network exceptions. Treat them as first-class application objects. See Recipe 1.5 for an example.