CHAPTER 7
Content Negotiation

Content negotiation, or *conneg* as it is sometimes called, is the process of selecting the best representation of a resource for a client when there are multiple representations (or *variants*) available. Although content negotiation is often associated with the practice of indicating media type preferences, content negotiation is also used to indicate preferences for localizing by language, character encoding, and compression.

HTTP specifies two types of content negotiation. These are server-driven negotiation and agent-driven negotiation. *Server-driven negotiation* uses request headers to select a variant, and *agent-driven negotiation* uses a distinct URI for each variant.

This chapter discusses the following recipes that deal with content negotiation:

Recipe 7.1, "How to Indicate Client Preferences"
: Use this recipe to decide which `Accept-*` headers to include when requesting a resource and with what values.

Recipe 7.2, "How to Implement Media Type Negotiation"
: Use this recipe to learn how to implement servers that correctly interpret the `Accept` request header for media type negotiation.

Recipe 7.3, "How to Implement Language Negotiation"
: Use this recipe to learn how to implement language negotiation using the `Accept-Language` header.

Recipe 7.4, "How to Implement Character Encoding Negotiation"
: Use this recipe to learn how to determine the requested character encoding for a representation.

Recipe 7.5, "How to Support Compression"
: HTTP allows clients to indicate their preference for compressed representations via the `Accept-Encoding` request header. Use this recipe to decide how to process this header on the server.

Recipe 7.6, "When and How to Send the Vary Header"
: Use this recipe to learn how to use the `Vary` header.

Recipe 7.7, "How to Handle Negotiation Failures"
: Use this recipe to determine when and how to return an error when the preferred variant is not available.

Recipe 7.8, "How to Use Agent-Driven Content Negotiation"
: Agent-driven negotiation is an alternative for a client to ask for a specific representation of a resource. Use this recipe to learn when and how to use this.

Recipe 7.9, "When to Support Server-Driven Negotiation"
: Use this recipe to learn the pros and cons of supporting multiple representations.

7.1 How to Indicate Client Preferences

When you are implementing a client, it is important for the client to indicate its preferences and capabilities to the server. These include representation formats it can process, languages it prefers, character encodings it can deal with, and its support for compression. Even when you know out of band the format, character encoding, language, and type of compression for a given representation in a response, clearly indicating the client's preferences and capabilities can help the client in the face of change. If not, when a server decides to offer an alternative representation for a resource, any default preferences your HTTP library may be using may prompt the server to return a different representation and break the client. It is better to ask for a specific representation instead of getting a default one, because the default representation can change.

Problem

You want to know how to allow a client to indicate its capabilities, such as supported media types, languages, etc.

Solution

When making a request, add an `Accept` header with a comma-separated list of media type preferences. If the client prefers one media type over the other, add a `q` parameter with each media type. This parameter indicates a relative preference for each media type listed in `Accept-*` headers. It is most commonly used with the `Accept` header. If the client can process only certain formats, add `*; q=0.0` in the `Accept` header to indicate to the server it cannot process anything other than the media types listed in the `Accept` header.

If the client can process characters of a specific character set only, add an `Accept-Charset` header with the preferred character set. If not, avoid adding this header.

Add an `Accept-Language` header for the preferred language of the representation.

If the client is able to decompress representations compressed using encodings such as `gzip`, `compress`, or `deflate`, add an `Accept-Encoding` header listing the supported encodings. If not, skip this header.

Discussion

In HTTP, the purpose of the `Accept-*` header is to let the client express its preferences for response representation. The server, based on its own capabilities, evaluates client preferences and determines an appropriate representation to return. Since the server determines the outcome of this process, this technique is called *server-driven negotiation*.

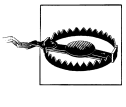
For example, consider a client with the following preferences:

- The client prefers a French representation but can accept English.
- The client can process an Atom-formatted representation with media type `application/atom+xml`, can accept an XML-formatted representation with media type `application/xml` representation, but cannot accept anything else.
- The client knows how to process `gzip`-compressed representations.

The client can indicate these preferences with the following request headers:

```
# Request headers
Accept: application/atom+xml;q=1.0, application/xml;q=0.6, */*;q=0.0
Accept-Language: fr;q=1.0, en;q=0.5
Accept-Encoding: gzip
```

In these headers, the part after the semicolon is the `q` parameter. The value of this header parameter is a floating-point number usually with one digit after the decimal, although HTTP 1.1 allows using up to three digits after the decimal. Clients can use this parameter to indicate the relative preference of each option in a range of `0.0` (i.e., unacceptable) to `1.0` (i.e., most preferred). For instance, the previous `Accept` header indicates that the client cannot process anything other than Atom- and XML-formatted representations. The default value of the `q` parameter is `1.0`.



Not all servers support `q` parameters. Such servers may select the first supported media type from the `Accept` header.

Note that servers may not always support content negotiation completely or correctly. Clients should be prepared to receive a representation that does not meet the `Accept-*` headers. Recipe 3.2 discusses how to use entity headers such as `Content-Type` to determine how to process response representations.

`Accept-*` headers such as `Accept` and `Accept-Language` express ranges of media types, languages, etc.

`Content-*` headers, on the other hand, express a specific media type, language, etc.

7.2 How to Implement Media Type Negotiation

Whether the server supports one media type or several media types for any resource, correctly interpreting the `Accept` header is necessary to improve interoperability.

Problem

You want to know how to decide which media type to use for a representation in a response.

Solution

If the request has no `Accept` header, return a representation using the default format for the requested resource.

If the request has an `Accept` header, parse the header and sort the values of media types by the `q` parameters in descending order. Then select a media type from the list that the server supports. Include a `Vary` response header as described in Recipe 7.6.

If the server does not support any of the media types in this list, use Recipe 7.7 to determine an appropriate response.

Discussion

Consider this `Accept` header:

```
Accept: application/atom+xml;q=1.0, application/xml;q=0.6, text/html
```

This includes two media types with different `q` parameter values and a third media type with no `q` parameter value. Since the default value of the `q` parameter is `1.0`, this header is equivalent to the following:

```
Accept: application/atom+xml;q=1.0, application/xml;q=0.6, text/html;q=1.0
```

From such a header, the server's first choice should be either `application/atom+xml` or `text/html`, and the second choice should be `application/xml`. If the server supports all three, it can return either Atom- or HTML-formatted representations. Given the value of the `Accept` header, either choice should be acceptable to the client:

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

... representation ...
```

Although implementing this logic is simple, certain situations can break interoperability with clients. Consider the following.

In the beginning, assume that the server supports `application/xml` for all its representations. Since it is not serving any other representation, it chooses to ignore the `Accept` header and returns XML-formatted representations for all requests. Then, because of client demand, it adds support for `application/json` and decides to rely on the value of the `Accept` header. This results in the following:

```
# Original request
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept: application/json

# Server can only support XML
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

... xml ...

# Same request now
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept: application/json

# Response - server supports JSON as well as XML
HTTP/1.1 200 OK
Content-Type: application/json

... json ...
```

This breaks compatibility because a working client no longer works. When you are faced with such situations, serve the new representation using a new URI. Here is an example:

```
# Same request now
GET /movie/gone_with_the_wind?format=json HTTP/1.1
Host: www.example.org
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

... json ...
```

This technique is called *agent-driven negotiation*. See Recipe 7.8.

7.3 How to Implement Language Negotiation

HTTP's support for language negotiation can help with limited localization support for web services. Language selection is just one aspect of localization. Apart from

translation of human-readable text in representations, localization often involves the regional and cultural adaptation of information.

Problem

You want to know how to decide the language to use for the human-readable text in a representation.

Solution

If the request has no `Accept-Language` header, return a representation with all human-readable text in a default language.

If the request has an `Accept-Language` header, parse the header, sort the media types by the `q` parameters, and select the first language in the list that the server can support. Include a `Vary` response header as described in Recipe 7.6.

If the server does not support any languages in the list and the `Accept-Language` header does not contain `*; q=0.0`, use a default language for that resource.

Discussion

The protocol for language negotiation is similar to media type negotiation. The client expresses its intent by supplying an `Accept-Language` header with acceptable languages and their `q` header parameter values, and the server decides which one to use for the response.

```
# Request
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept-Language: en,en-US,fr;q=0.6

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

<movie>
  <title>Gone with the Wind</title>
  <year>1936</year>
  ...
</movie>
```

This approach is best suited when representations in different languages differ only in terms of the language used for any human-readable text in the representation, as in the following representation:

```
# Request
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept-Language: en,en-US,fr;q=0.6
```

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

<movie>
  <title>Autant en emporte le vent</title>
  <year>1936</year>
  ...
</movie>
```

If the differences between representations are more significant, use other means of localization such as the client's IP address or region/language-specific URIs.

7.4 How to Implement Character Encoding Negotiation

If the client asks (via the `Accept-Charset` header) for textual representations to be encoded in a particular character encoding, encoding the response using that encoding promotes interoperability.

Problem

You want to know what character encoding to use for textual representations in responses.

Solution

If the request has no `Accept-Charset` header, return a representation using `UTF-8` encoding.

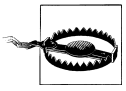
If the request has an `Accept-Charset` header, parse the header, sort the character sets by the `q` parameters, and select the character set that the server can support for encoding.

If the server does not support any requested character sets and the `Accept-Charset` header does not contain `*; q=0.0`, return a representation using `UTF-8` encoding.

In all these cases, if the media type is textual and allows a `charset` parameter, include the `charset` parameter in the `Content-Type` header indicating the character encoding that the server used. Also include a `Vary` response header as described in Recipe 7.6.

Discussion

Most platforms and programming languages support `UTF-8`. `UTF-8` is the default encoding for both `application/xml` and `application/json` media types. Always use `UTF-8` encoding unless the client asks for a different encoding.



Avoid using `text/xml` since its default encoding is `US-ASCII`.

7.5 How to Support Compression

Servers can optionally serve compressed representations to clients using encodings such as `gzip`, `deflate`, or `compress`. In HTTP, this technique is generally called *content encoding*.

Problem

You want to know when to enable compression of representations.

Solution

If the server is capable of compressing-response body, select the compression technique from the `Accept-Encoding` header. Include a `Vary` response header as described in Recipe 7.6. If no encoding in this header matches the server's supported encodings, ignore this header. The `q` parameter processing is similar to other `Accept-*` headers.

If the request has no `Accept-Encoding` header, do not compress representations.

Discussion

Clients may or may not support content encodings such as `gzip` or `deflate`. It is important to return compressed responses only when the client sends an `Accept-Encoding` header with a compression format that the server supports. Here is an example:

```
# Request
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept-Encoding: gzip

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Content-Encoding: gzip
Vary: Accept-Encoding

... gzipped bytes ...
```

In most cases, you should be able to configure your HTTP servers to automatically apply a given encoding for responses. For instance, placing the following line in your Apache HTTP server configuration tells the server to apply `deflate` encoding to all `application/xml` representations:

```
AddOutputFilterByType DEFLATE application/xml
```

7.6 When and How to Send the Vary Header

When a server uses content negotiation to select a representation, the same URI can yield different representations based on `Accept-*` headers. The `Vary` header tells clients which request headers the server used when selecting a representation.

Problem

You want to know how to use the `Vary` header to indicate clients how the server chose a particular representation.

Solution

Include a `Vary` header whenever multiple representations are available for a resource. The value of this header is a comma-separated list of request headers the server uses when choosing a representation. If the server uses information other than the headers in the request, such as the client's IP address, time of the day, user personalization, etc., include a `Vary` header with a value of `*`.

Discussion

The server can use the `Vary` header to inform clients of the results of server-driven content negotiation. The value of the `Vary` header is a set of request headers and not response headers. For instance, consider the following sequences of requests and responses:

```
# Request for English representation
GET /status HTTP/1.1
Host: www.example.org
Accept-Language: en;q=1.0,*/*;q=0.0

# Response
HTTP/1.1 200 OK
Content-Language: en
Vary: Accept-Language

...

# Request for German representation
GET /status HTTP/1.1
Host: www.example.org
Accept-Language: de;q=1.0,*/*;q=0.0

# Response
HTTP/1.1 200 OK
Content-Language: de
Vary: Accept-Language

...

# Request for French representation
```

```
GET /status HTTP/1.1
Host: www.example.org
Accept-Language: fr;q=1.0,*/*;q=0.0

# Response
HTTP/1.1 200 OK
Content-Language: fr
Vary: Accept-Language
```

Although the request URI is the same, clients and intermediaries can differentiate between the responses by looking at the value of the request headers listed in `Vary` header. Caches use this header as part of cache keys to maintain variants of a resource. Clients can use this information to know the criteria the server used for content negotiation.

7.7 How to Handle Negotiation Failures

Servers are free to serve any available representation for a given resource. However, clients may not be able to handle arbitrary media types. Except for browsers, most HTTP clients can deal with only one or two formats.

Problem

You want to know whether to serve a default representation, or return an error, when the server is unable to serve a representation preferred by the client.

Solution

When the server cannot serve a representation that meets the client's preferences and if the client explicitly included a `*/*;q=0.0`, return status code `406` (`Not Acceptable`) with the body of the representation containing the list of representations.

If the server is unable to support requested `Accept-Encoding` values, serve the representation without applying any content encoding.

Discussion

Here is an example of a request from a client that can process no media type except `application/json`:

```
# Request
GET /user/001/followers HTTP/1.1
Accept: application/json,*/*;q=0.0 ❶

# Response
406 Not Acceptable ❷
Content-Type: application/json
Link: <http://www.example.org/errors/mediatypes.html>;rel="help" ❸

{
  "message" : "This server does not support JSON. See help for alternatives."
}
```

❶ The client cannot process anything other than JSON.

❷ The server does not support JSON.

❸ A link with help on supported formats.

In this example, the server recognizes JSON but is unable to serve a representation of the resource in that format. Since the client request includes a `q=0.0` for every other media type except `application/json`, a failure is acceptable for the client.

Note that the server uses a JSON-formatted representation for the error message. It is quite reasonable for the server to implement error messages in commonly used formats. If not, return the error message in human-readable HTML format.

```
# Request
GET /user/001/followers HTTP/1.1
Accept: application/json,*/*;q=0.0

# Response
406 Not Acceptable
Content-Type: text/html;charset=UTF-8
Link: <http://www.example.org/errors/mediatypes.html>;rel="help"

<html>
  <head>
    <title>JSON Not Supported</title>
  </head>
  <body>
    <p>This server does not support JSON. See <a
    href="http://www.example.org/errors/mediatypes.html">help</a> for alternatives.</p>
  </body>
</html>
```

7.8 How to Use Agent-Driven Content Negotiation

Although server-driven negotiation is built into HTTP, it has limitations:

- Content negotiation does not include elements such as currency units, distance units, date formats, and other regional flavors for any human-readable text in representations. For instance, you cannot always determine currency and date formats based on the language preference.
- In some cases, because of complex localization requirements, the server may decide to maintain different resources for different locales.
- Common web browsers use a broad range of media types for the `Accept` headers. For instance, some installations of the Firefox browser send `Accept: text/html, application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`. This makes it difficult to view content-negotiated representations in browsers.

For these, use agent-driven negotiation. Agent-driven negotiation is useful when the client cannot communicate its preferences using `Accept-*` headers.

Problem

You want to know how to implement agent-driven negotiation.

Solution

Provide a URI for each representation.

Discussion

Agent-driven negotiation simply means providing a distinct URI for each variant and allowing the client to use that URI to select the desired representation. In agent-driven negotiation, client uses out-of-band information from the server to determine which URI to use. If the representation exists, the server returns it. If it does not, it returns a `404` (`Not Found`) response code.

Since the client determines the outcome of this process, this technique is called *agent-driven negotiation*. The term *agent* refers to user agents, and the most common user agents are browsers.

Although it is possible to implement agent-driven negotiation for all `Accept-*` headers, in practice it is most commonly used for media types and languages.

There are several ways for the server to assign URIs for each language and media type of a resource. Some commonly used approaches include the following:

Query parameters
: Append the language and or media type as query parameters to a base URI, with the values of these query parameters using a shorthand notation for media types. Examples include `format` for language negotiation and `lang` to support media type negotiation. Here are some examples:

```
http://www.example.org/status?format=json
http://www.example.org/status?format=xml
http://www.example.org/status?format=csv
```

URI extensions
: Append a dot (the `.` character), and shorthand media type to a base URI. Examples include `status.atom` for an `application/atom+xml` representation and `status.json` for an `application/json` representation.

Subdomains
: Create subdomains to support language-specific representations. Examples include `en.wikipedia.org` to serve English-language representations of Wikipedia entries and `de.wikipedia.org` to serve German-language representations of Wikipedia entries.

When using agent-driven negotiation, the server can choose to advertise alternatives using links with the `alternate` link relation type. Here is an example:

```
<status xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/status?format=xml&lang=en"/> ❶
  <atom:link rel="alternate" type="application/json"
        href="http://www.example.org/status?format=json&lang=fr"/> ❷
  ...
</status>
```

❶ A link to the resource

❷ A link to a variant

7.9 When to Support Server-Driven Negotiation

Content negotiation is not always appropriate. Although some popular web services and web service frameworks support general-purpose formats such as XML, JSON, and Atom for every resource, consider the cost of supporting multiple formats in your web services.

Problem

You want to know if server-driven negotiation is right for your web service.

Solution

Support multiple variants when only your clients need them or whether each variant contains the same information. If the information content is different, use a distinct URI for each.

Discussion

Content negotiation is only cheap to implement when your development framework supports it. In other cases, content negotiation takes time and effort to implement, test, and manage. Most client applications can deal only with a single format. In such cases, supporting multiple formats may be unnecessary. Before deciding to support multiple representations for each resource, consider the following:

- In some cases, the application flow may be different for each representation format. This is particularly true for HTML representations. User interface constraints may require HTML representations to follow a different application flow from the one used for, say, XML-formatted representations. In this case, server-driven negotiation for both HTML and XML formats is not realistic.
- Unambiguously returning a variant based on the `Accept` header with several media types with different `q` is not trivial. Not all development frameworks support this.

- Language negotiation may be simplistic for global services. In some cases, legal and business requirements may be regional, and agent-driven negotiation may be the best approach.
- Caches may not handle content-negotiated responses well. Some caches may ignore or limit the number of variants they store for any given resource.

Given these, carefully consider the requirements before supporting server-driven content negotiation in your server applications.