HTTP

Succinctly

by Scott Allen

Syncfusion | Technology Resource Portal

Chapter 1 Resources

Perhaps the most familiar part of the web is the HTTP address. When I want to find a recipe for a dish featuring broccoli, which is almost never, then I might open my web browser and enter `http://food.com` in the address bar to go to the food.com website and search for recipes. My web browser understands this syntax and knows it needs to make an HTTP request to a server named food.com. We'll talk later about what it means to "make an HTTP request" and all the networking details involved. For now, we just want to focus on the address: `http://food.com`.

Resource Locators

The address `http://food.com` is what we call a URL—a uniform resource locator. It represents a specific resource on the web. In this case, the resource is the home page of the food.com website. Resources are things I want to interact with on the web. Images, pages, files, and videos are all resources.

There are billions, if not trillions, of places to go on the Internet—in other words, there are trillions of resources. Each resource will have a URL I can use to find it. `http://news.google.com` is a different place than `http://news.yahoo.com`. These are two different names, two different companies, two different websites, and therefore two different URLs. Of course, there will also be different URLs inside the same website. `http://food.com/recipe/broccoli-salad-10733/` is the URL for a page with a broccoli salad recipe, while `http://food.com/recipe/grilled-cauliflower-19710/` is still at food.com, but is a different resource describing a cauliflower recipe.

We can break the last URL into three parts:

1. `http`, the part before the `://`, is what we call the **URL scheme**. The scheme describes *how* to access a particular resource, and in this case it tells the browser to use the hypertext transfer protocol. Later we'll also look at a different scheme, HTTPS, which is the secure HTTP protocol. You might run into other schemes too, like FTP for the file transfer protocol, and mailto for email addresses.

   Everything after the `://` will be specific to a particular scheme. So, a legal HTTP URL may not be a legal mailto URL—those two aren't really interchangeable (which makes sense because they describe different types of resources).

2. `food.com` is the **host**. This host name tells the browser the name of the computer hosting the resource. The computer will use the Domain Name System (DNS) to translate `food.com` into a network address, and then it will know exactly where to send the request for the resource. You can also specify the host portion of a URL using an IP address.

3. `/recipe/grilled-cauliflower-19710/` is the **URL path**. The food.com host should recognize the specific resource being requested by this path and respond appropriately.

Sometimes a URL will point to a file on the host's file system or hard drive. For example, the URL `http://food.com/logo.jpg` might point to a picture that really does exist on the

food.com server. However, resources can also be dynamic. The URL `http://food.com/recipes/brocolli` probably does not refer to a real file on the food.com server. Instead, some sort of application is running on the food.com host that will take that request and build a resource using content from a database. The application might be built using ASP.NET, PHP, Perl, Ruby on Rails, or some other web technology that knows how to respond to incoming requests by creating HTML for a browser to display.

In fact, these days many websites try to *avoid* having any sort of real file name in their URL. For starters, file names are usually associated with a specific technology, like .aspx for Microsoft's ASP.NET technology. Many URLs will outlive the technology used to host and serve them. Secondly, many sites want to place keywords into a URL (like having `/recipe/broccoli/` in the URL for a broccoli recipe). Having these keywords in the URL is a form of search engine optimization (SEO) that will rank the resource higher in search engine results. Descriptive keywords, not file names, are important for URLs these days.

Some resources will also lead the browser to download additional resources. The food.com home page will include images, JavaScript files, CSS, and other resources that will all combine to present the "home page" of food.com.

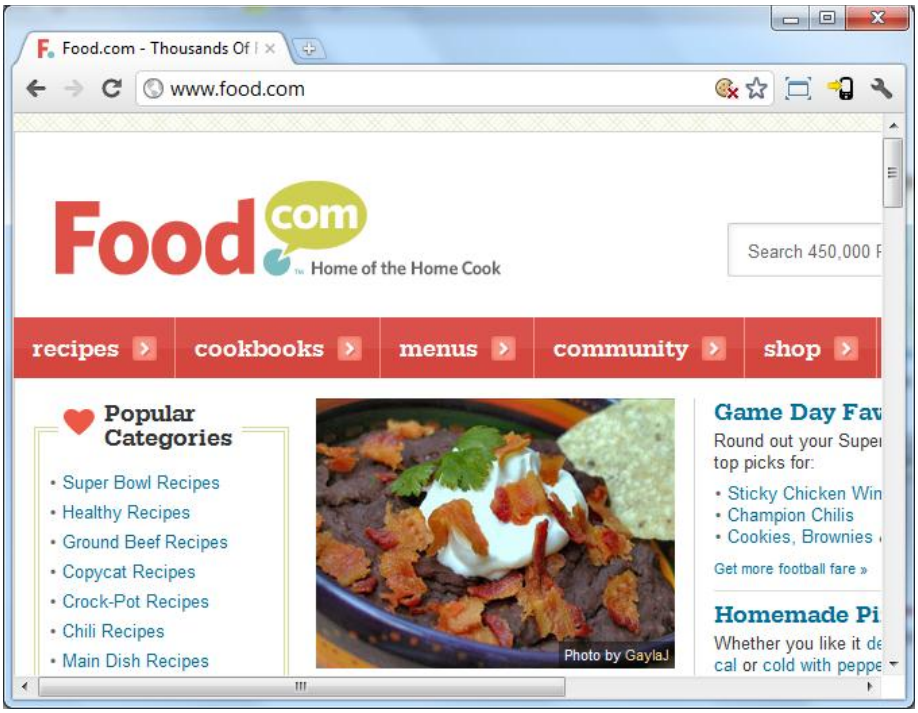

Figure 1: food.com home page

Ports, Query Strings, and Fragments

Now that we know about URL schemes, hosts, and paths, let's also look at a URL with a port number:

```
http://food.com:80/recipes/broccoli/
```

The number 80 represents the **port number** the host is using to listen for HTTP requests. The default port number for HTTP is port 80, so you generally see this port number omitted from a URL. You only need to specify a port number if the server is listening on a port other than port 80, which usually only happens in testing, debugging, or development environments. Let's look at another URL.

```
http://www.bing.com/search?q=broccoli
```

Everything after `?` (the question mark) is known as the **query**. The query, also called the **query string**, contains information for the destination website to use or interpret. There is no formal standard for how the query string should look as it is technically up to the application to interpret the values it finds, but you'll see the majority of query strings used to pass name–value pairs in the form `name1=value1&name2=value2`.

For example:

```
http://foo.com?first=Scott&last=Allen
```

There are two name–value pairs in this example. The first pair has the name "first" and the value "Scott". The second pair has the name "last" with the value "Allen". In our earlier URL (`http://www.bing.com/search?q=broccoli`), the Bing search engine will see the name "q" associated with the value "broccoli". It turns out the Bing engine looks for a “q” value to use as the search term. We can think of the URL as the URL for the resource that represents the Bing search results for broccoli.

Finally, one more URL:

```
http://server.com?recipe=broccoli#ingredients
```

The part after the # sign is known as the **fragment**. The fragment is different than the other pieces we've looked at so far, because unlike the URL path and query string, the fragment is not processed by the server. The fragment is only used on the client and it identifies a particular section of a resource. Specifically, the fragment is typically used to identify a specific HTML element in a page by the element's ID.

Web browsers will typically align the initial display of a webpage such that the top of the element identified by the fragment is at the top of the screen. As an example, the URL `http://odetocode.com/Blogs/scott/archive/2011/11/29/programming-windows-8-the-sublime-to-the-strange.aspx#feedback` has the fragment value "feedback". If you follow the URL, your web browser should scroll down the page to show the feedback section of a particular blog post on my blog. Your browser retrieved the entire resource (the blog post), but focused your attention to a specific area—the feedback section. You can imagine the HTML for the blog post looking like the following (with all the text content omitted):

```
<div id="post">
    ...
</div>
<div id="feedback">
    ...
</div>
```

The client makes sure the element with the “feedback” ID is at the top.

If we put together everything we've learned so far, we know a URL is broken into the following pieces:

```
<scheme>://<host>:<port>/<path>?<query>#<fragment>
```

URL Encoding

All software developers who work with the web should be aware of character encoding issues with URLs. The official documents describing URLs go to great lengths to make URLs as usable and interoperable as possible. A URL should be as easy to communicate through email as it is to print on a bumper sticker and affix to a 2001 Ford Windstar. For this reason, the Internet standards define **unsafe characters** for URLs. For example, the space character is considered unsafe because space characters can mistakenly appear or disappear when a URL is in printed form (is that one space or two spaces on your business card?).

Other unsafe characters include the number sign (#) because it is used to delimit a fragment, and the caret (^) because it isn't always transmitted correctly through all network devices. In fact, RFC 3986 (the "law" for URLs), defines the safe characters for URLs to be the alphanumeric characters in US-ASCII, plus a few special characters like the colon (:) and the slash mark (/).

Fortunately, you can still transmit unsafe characters in a URL, but all unsafe characters must be percent-encoded (aka URL encoded). `%20` is the encoding for a space character (where 20 is the hexadecimal value for the US-ASCII space character).

As an example, let's say you wanted to create the URL for a file named "^my resume.txt" on someserver.com. The legal, encoded URL would look like:

```
http://someserver.com/%5Emy%20resume.txt
```

Both the ^ and space characters have been percent-encoded. Most web application frameworks will provide an API for easy URL encoding. On the server side, you should run your dynamically created URLs through an encoding API just in case one of the unsafe characters appears in the URL.

Resources and Media Types

So far we've focused on URLs and simplified everything else. But, what does it mean when we enter a URL into the browser? Typically it means we want to retrieve or view some resource. There is a tremendous amount of material to view on the web, and later we'll also see how HTTP also enables us to create, delete, and update resources. For now, we'll stay focused on retrieval.

We haven't been very specific about the types of resources we want to retrieve. There are thousands of different resource types on the web—images, hypertext documents, XML documents, video, audio, executable applications, Microsoft Word documents, and countless more.

In order for a host to properly serve a resource, and in order for a client to properly display a resource, the parties involved have to be specific and precise about the type of the resource. Is the resource an image? Is the resource a movie? We wouldn't want our web browsers to try rendering a PNG image as text, and we wouldn't want them to try interpreting hypertext as an image.

When a host responds to an HTTP request, it returns a resource and also specifies the **content type** (also known as the media type) of the resource. We'll see the details of how the content type appears in an HTTP message in the next chapter.

To specify content types, HTTP relies on the Multipurpose Internet Mail Extensions (MIME) standards. Although MIME was originally designed for email communications, HTTP uses MIME standards for the same purpose, which is to label the content in such a way that the client will know what the content contains.

For example, when a client requests an HTML webpage, the host can respond to the HTTP request with some HTML that it labels as "`text/html`". The "`text`" part is the primary media type, and the "`html`" is the media subtype. When responding to the request for an image, the host will label the resource with a content type of "`image/jpeg`" for JPG files, "`image/gif`" for GIF files, or "`image/png`" for PNG files. Those content types are standard MIME types and are literally what will appear in the HTTP response.

A Quick Note on File Extensions

You might think that a browser would rely on the file extension to determine the content type of an incoming resource. For example, if my browser requests "frog.jpg" it should treat the resource as a JPG file, but treat "frog.gif" as a GIF file. However, for most browsers, the file extension is the last place it will go to determine the actual content type.

File extensions can be misleading, and just because we requested a JPG file doesn't mean the server has to respond with data encoded in JPG format. Microsoft documents Internet Explorer (IE) as first looking at the content type tag specified by the host. If the host doesn't provide a content type, IE will then scan the first 200 bytes of the response trying to guess the content type. Finally, if IE doesn't find a content type and can't guess the content type, it will fall back on the file extension used in the request for the resource. This is one reason why the content type label is important, but it is far from the only reason.

Content Type Negotiation

Although we tend to think of HTTP as something used to serve webpages, it turns out the HTTP specification describes a flexible, generic protocol for moving high-fidelity information. Part of the job of moving information around is making sure all the parties involved know how to interpret the information, and this is why the media type settings are important.

However, media types aren't just for hosts. Clients can play a role in what media type a host returns by taking part in a content type negotiation.

A resource identified by a single URL can have **multiple representations**. Take, for example, the broccoli recipe we mentioned earlier. The single recipe might have representations in different languages (English, French, and German). The recipe could even have representations in different formats (HTML, PDF, and plain text). It's all the same resource and the same recipe, but different representations.

The obvious question is: Which representation should the server select? The answer is in the content negotiation mechanism described by the HTTP specification. When a client makes an HTTP request to a URL, the client can specify the media types it will accept. The media types are not only for the host to tag outgoing resources, but also for clients to specify the media type they want to consume.

The client specifies what it will accept in the outgoing request message. Again, we'll see details of this message in Chapter 2, but imagine a request to `http://food.com/` saying it will accept a representation in the German language. It's up to the server to try fulfilling the request. The host might send a textual resource that is still in English, which will probably disappoint a German-speaking user, but this is why we call it content negotiation and not content ultimatum.

Web browsers are sophisticated pieces of software that can deal with many different types of resource representations. Content negotiation is something a user would probably never care about, but for software developers (especially web service developers) content negotiation is part of what makes HTTP great. A piece of code written in JavaScript can make a request to the server and ask for a JSON representation. A piece of code written in C++ can make a request to the server and ask for an XML representation. In both cases, if the host can satisfy the request, the information will arrive at the client in an ideal format for parsing and consumption.

Where Are We?

At this point we've gotten about as far as we can go without getting into the nitty-gritty details of what an HTTP message looks like. We've learned about URLs, URL encoding, and content types. It's time to see what these content type specifications look like as they travel across the wire.

HTTP

Succinctly

by Scott Allen

Syncfusion | Technology Resource Portal

Chapter 1 Resources

Perhaps the most familiar part of the web is the HTTP address. When I want to find a recipe for a dish featuring broccoli, which is almost never, then I might open my web browser and enter `http://food.com` in the address bar to go to the food.com website and search for recipes. My web browser understands this syntax and knows it needs to make an HTTP request to a server named food.com. We'll talk later about what it means to "make an HTTP request" and all the networking details involved. For now, we just want to focus on the address: `http://food.com`.

Resource Locators

The address `http://food.com` is what we call a URL—a uniform resource locator. It represents a specific resource on the web. In this case, the resource is the home page of the food.com website. Resources are things I want to interact with on the web. Images, pages, files, and videos are all resources.

There are billions, if not trillions, of places to go on the Internet—in other words, there are trillions of resources. Each resource will have a URL I can use to find it. `http://news.google.com` is a different place than `http://news.yahoo.com`. These are two different names, two different companies, two different websites, and therefore two different URLs. Of course, there will also be different URLs inside the same website. `http://food.com/recipe/broccoli-salad-10733/` is the URL for a page with a broccoli salad recipe, while `http://food.com/recipe/grilled-cauliflower-19710/` is still at food.com, but is a different resource describing a cauliflower recipe.

We can break the last URL into three parts:

1. `http`, the part before the `://`, is what we call the **URL scheme**. The scheme describes *how* to access a particular resource, and in this case it tells the browser to use the hypertext transfer protocol. Later we'll also look at a different scheme, HTTPS, which is the secure HTTP protocol. You might run into other schemes too, like FTP for the file transfer protocol, and mailto for email addresses.

   Everything after the `://` will be specific to a particular scheme. So, a legal HTTP URL may not be a legal mailto URL—those two aren't really interchangeable (which makes sense because they describe different types of resources).

2. `food.com` is the **host**. This host name tells the browser the name of the computer hosting the resource. The computer will use the Domain Name System (DNS) to translate `food.com` into a network address, and then it will know exactly where to send the request for the resource. You can also specify the host portion of a URL using an IP address.

3. `/recipe/grilled-cauliflower-19710/` is the **URL path**. The food.com host should recognize the specific resource being requested by this path and respond appropriately.

Sometimes a URL will point to a file on the host's file system or hard drive. For example, the URL `http://food.com/logo.jpg` might point to a picture that really does exist on the

food.com server. However, resources can also be dynamic. The URL `http://food.com/recipes/brocolli` probably does not refer to a real file on the food.com server. Instead, some sort of application is running on the food.com host that will take that request and build a resource using content from a database. The application might be built using ASP.NET, PHP, Perl, Ruby on Rails, or some other web technology that knows how to respond to incoming requests by creating HTML for a browser to display.

In fact, these days many websites try to *avoid* having any sort of real file name in their URL. For starters, file names are usually associated with a specific technology, like .aspx for Microsoft's ASP.NET technology. Many URLs will outlive the technology used to host and serve them. Secondly, many sites want to place keywords into a URL (like having `/recipe/broccoli/` in the URL for a broccoli recipe). Having these keywords in the URL is a form of search engine optimization (SEO) that will rank the resource higher in search engine results. Descriptive keywords, not file names, are important for URLs these days.

Some resources will also lead the browser to download additional resources. The food.com home page will include images, JavaScript files, CSS, and other resources that will all combine to present the "home page" of food.com.

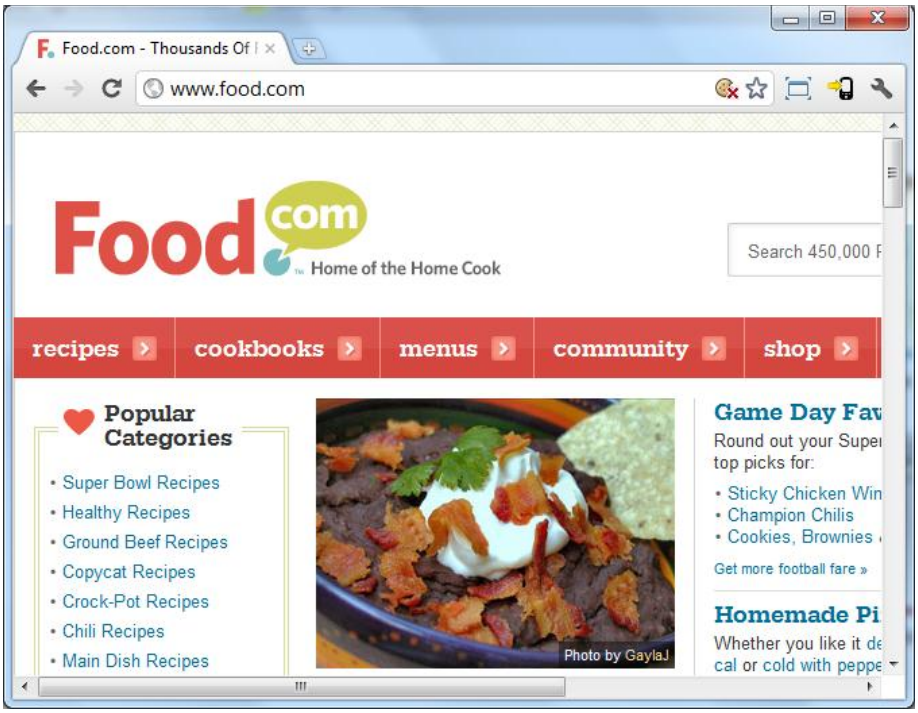

Figure 1: food.com home page

Ports, Query Strings, and Fragments

Now that we know about URL schemes, hosts, and paths, let's also look at a URL with a port number:

```
http://food.com:80/recipes/broccoli/
```

The number 80 represents the **port number** the host is using to listen for HTTP requests. The default port number for HTTP is port 80, so you generally see this port number omitted from a URL. You only need to specify a port number if the server is listening on a port other than port 80, which usually only happens in testing, debugging, or development environments. Let's look at another URL.

```
http://www.bing.com/search?q=broccoli
```

Everything after `?` (the question mark) is known as the **query**. The query, also called the **query string**, contains information for the destination website to use or interpret. There is no formal standard for how the query string should look as it is technically up to the application to interpret the values it finds, but you'll see the majority of query strings used to pass name–value pairs in the form `name1=value1&name2=value2`.

For example:

```
http://foo.com?first=Scott&last=Allen
```

There are two name–value pairs in this example. The first pair has the name "first" and the value "Scott". The second pair has the name "last" with the value "Allen". In our earlier URL (`http://www.bing.com/search?q=broccoli`), the Bing search engine will see the name "q" associated with the value "broccoli". It turns out the Bing engine looks for a “q” value to use as the search term. We can think of the URL as the URL for the resource that represents the Bing search results for broccoli.

Finally, one more URL:

```
http://server.com?recipe=broccoli#ingredients
```

The part after the # sign is known as the **fragment**. The fragment is different than the other pieces we've looked at so far, because unlike the URL path and query string, the fragment is not processed by the server. The fragment is only used on the client and it identifies a particular section of a resource. Specifically, the fragment is typically used to identify a specific HTML element in a page by the element's ID.

Web browsers will typically align the initial display of a webpage such that the top of the element identified by the fragment is at the top of the screen. As an example, the URL `http://odetocode.com/Blogs/scott/archive/2011/11/29/programming-windows-8-the-sublime-to-the-strange.aspx#feedback` has the fragment value "feedback". If you follow the URL, your web browser should scroll down the page to show the feedback section of a particular blog post on my blog. Your browser retrieved the entire resource (the blog post), but focused your attention to a specific area—the feedback section. You can imagine the HTML for the blog post looking like the following (with all the text content omitted):

```
<div id="post">
    ...
</div>
<div id="feedback">
    ...
</div>
```

The client makes sure the element with the “feedback” ID is at the top.

If we put together everything we've learned so far, we know a URL is broken into the following pieces:

```
<scheme>://<host>:<port>/<path>?<query>#<fragment>
```

URL Encoding

All software developers who work with the web should be aware of character encoding issues with URLs. The official documents describing URLs go to great lengths to make URLs as usable and interoperable as possible. A URL should be as easy to communicate through email as it is to print on a bumper sticker and affix to a 2001 Ford Windstar. For this reason, the Internet standards define **unsafe characters** for URLs. For example, the space character is considered unsafe because space characters can mistakenly appear or disappear when a URL is in printed form (is that one space or two spaces on your business card?).

Other unsafe characters include the number sign (#) because it is used to delimit a fragment, and the caret (^) because it isn't always transmitted correctly through all network devices. In fact, RFC 3986 (the "law" for URLs), defines the safe characters for URLs to be the alphanumeric characters in US-ASCII, plus a few special characters like the colon (:) and the slash mark (/).

Fortunately, you can still transmit unsafe characters in a URL, but all unsafe characters must be percent-encoded (aka URL encoded). `%20` is the encoding for a space character (where 20 is the hexadecimal value for the US-ASCII space character).

As an example, let's say you wanted to create the URL for a file named "^my resume.txt" on someserver.com. The legal, encoded URL would look like:

```
http://someserver.com/%5Emy%20resume.txt
```

Both the ^ and space characters have been percent-encoded. Most web application frameworks will provide an API for easy URL encoding. On the server side, you should run your dynamically created URLs through an encoding API just in case one of the unsafe characters appears in the URL.

Resources and Media Types

So far we've focused on URLs and simplified everything else. But, what does it mean when we enter a URL into the browser? Typically it means we want to retrieve or view some resource. There is a tremendous amount of material to view on the web, and later we'll also see how HTTP also enables us to create, delete, and update resources. For now, we'll stay focused on retrieval.

We haven't been very specific about the types of resources we want to retrieve. There are thousands of different resource types on the web—images, hypertext documents, XML documents, video, audio, executable applications, Microsoft Word documents, and countless more.

In order for a host to properly serve a resource, and in order for a client to properly display a resource, the parties involved have to be specific and precise about the type of the resource. Is the resource an image? Is the resource a movie? We wouldn't want our web browsers to try rendering a PNG image as text, and we wouldn't want them to try interpreting hypertext as an image.

When a host responds to an HTTP request, it returns a resource and also specifies the **content type** (also known as the media type) of the resource. We'll see the details of how the content type appears in an HTTP message in the next chapter.

To specify content types, HTTP relies on the Multipurpose Internet Mail Extensions (MIME) standards. Although MIME was originally designed for email communications, HTTP uses MIME standards for the same purpose, which is to label the content in such a way that the client will know what the content contains.

For example, when a client requests an HTML webpage, the host can respond to the HTTP request with some HTML that it labels as "`text/html`". The "`text`" part is the primary media type, and the "`html`" is the media subtype. When responding to the request for an image, the host will label the resource with a content type of "`image/jpeg`" for JPG files, "`image/gif`" for GIF files, or "`image/png`" for PNG files. Those content types are standard MIME types and are literally what will appear in the HTTP response.

A Quick Note on File Extensions

You might think that a browser would rely on the file extension to determine the content type of an incoming resource. For example, if my browser requests "frog.jpg" it should treat the resource as a JPG file, but treat "frog.gif" as a GIF file. However, for most browsers, the file extension is the last place it will go to determine the actual content type.

File extensions can be misleading, and just because we requested a JPG file doesn't mean the server has to respond with data encoded in JPG format. Microsoft documents Internet Explorer (IE) as first looking at the content type tag specified by the host. If the host doesn't provide a content type, IE will then scan the first 200 bytes of the response trying to guess the content type. Finally, if IE doesn't find a content type and can't guess the content type, it will fall back on the file extension used in the request for the resource. This is one reason why the content type label is important, but it is far from the only reason.

Content Type Negotiation

Although we tend to think of HTTP as something used to serve webpages, it turns out the HTTP specification describes a flexible, generic protocol for moving high-fidelity information. Part of the job of moving information around is making sure all the parties involved know how to interpret the information, and this is why the media type settings are important.

However, media types aren't just for hosts. Clients can play a role in what media type a host returns by taking part in a content type negotiation.

A resource identified by a single URL can have **multiple representations**. Take, for example, the broccoli recipe we mentioned earlier. The single recipe might have representations in different languages (English, French, and German). The recipe could even have representations in different formats (HTML, PDF, and plain text). It's all the same resource and the same recipe, but different representations.

The obvious question is: Which representation should the server select? The answer is in the content negotiation mechanism described by the HTTP specification. When a client makes an HTTP request to a URL, the client can specify the media types it will accept. The media types are not only for the host to tag outgoing resources, but also for clients to specify the media type they want to consume.

The client specifies what it will accept in the outgoing request message. Again, we'll see details of this message in Chapter 2, but imagine a request to `http://food.com/` saying it will accept a representation in the German language. It's up to the server to try fulfilling the request. The host might send a textual resource that is still in English, which will probably disappoint a German-speaking user, but this is why we call it content negotiation and not content ultimatum.

Web browsers are sophisticated pieces of software that can deal with many different types of resource representations. Content negotiation is something a user would probably never care about, but for software developers (especially web service developers) content negotiation is part of what makes HTTP great. A piece of code written in JavaScript can make a request to the server and ask for a JSON representation. A piece of code written in C++ can make a request to the server and ask for an XML representation. In both cases, if the host can satisfy the request, the information will arrive at the client in an ideal format for parsing and consumption.

Where Are We?

At this point we've gotten about as far as we can go without getting into the nitty-gritty details of what an HTTP message looks like. We've learned about URLs, URL encoding, and content types. It's time to see what these content type specifications look like as they travel across the wire.