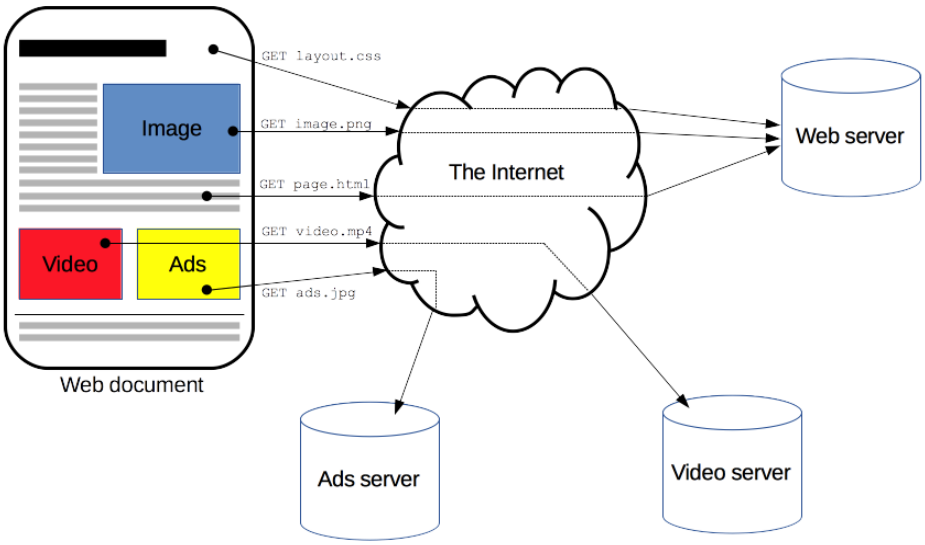
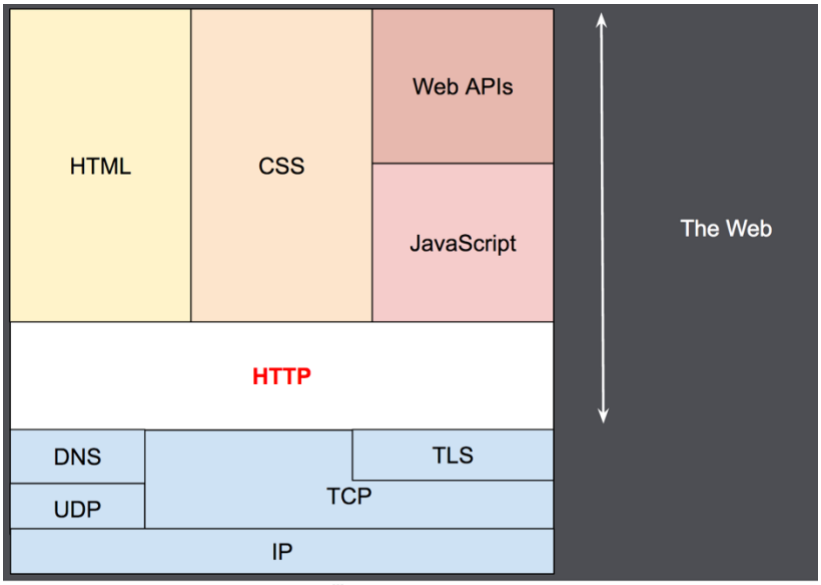
UNIDAD 7: PROTOCOLO HTTP Y PATRÓN MCV

Índice

1. PROTOCOLO HTTP

HTTP, de sus siglas en inglés: "Hypertext Transfer Protocol", es el nombre de un protocolo el cual nos permite realizar una petición de datos y recursos, como pueden ser documentos HTML. Es la base de cualquier intercambio de datos en la Web, y un protocolo de estructura cliente-servidor, esto quiere decir que una petición de datos es iniciada por el elemento que recibirá los datos (el cliente), normalmente un navegador Web. Así, una página web completa resulta de la unión de distintos sub-documentos recibidos, como, por ejemplo: un documento que especifique el estilo de maquetación de la página web (CSS), el texto, las imágenes, vídeos, scripts, etc...





Clientes y servidores se comunican intercambiando mensajes individuales (en contraposición a las comunicaciones que utilizan flujos continuos de datos). Los mensajes que envía el cliente, normalmente un navegador Web, se llaman *peticiones*, y los mensajes enviados por el servidor se llaman *respuestas*.

Diseñado a principios de la década de 1990, HTTP es un protocolo ampliable, que ha ido evolucionando con el tiempo. Es lo que se conoce como un protocolo de la capa de aplicación, y se transmite sobre el protocolo TCP, o el protocolo encriptado TLS, aunque teóricamente podría usarse cualquier otro protocolo fiable. Gracias a que es un protocolo capaz de ampliarse, se usa no solo para transmitir documentos de hipertexto (HTML), sino que además, se usa para transmitir imágenes o vídeos, o enviar datos o contenido a los servidores, como en el caso de los formularios de datos. HTTP puede incluso ser utilizado para transmitir partes de documentos, y actualizar páginas Web en el acto.

Arquitectura de los sistemas basados en HTTP

HTTP es un protocolo basado en el principio de cliente-servidor: las peticiones son enviadas por una entidad: el agente del usuario (o un proxy a petición de uno). La mayoría de las veces el agente del usuario (cliente) es un navegador Web, pero podría ser cualquier otro programa, como por ejemplo un programa-robot, que explore la Web, para adquirir datos de su estructura y contenido para uso de un buscador de Internet.

Cada petición individual se envía a un servidor, el cual la gestiona y responde. Entre cada *petición* y *respuesta*, hay varios intermediarios, normalmente denominados proxies, los cuales realizan distintas funciones, como: gateways o caches.

En realidad, hay más elementos intermedios, entre un navegador y el servidor que gestiona su petición: hay otros tipos de dispositivos: como routers, modems ... Es gracias a la arquitectura en capas de la Web, que estos intermediarios, son transparentes al navegador y al servidor, ya que HTTP se apoya en los protocolos de red y transporte. HTTP es un protocolo de aplicación, y por tanto se apoya sobre los anteriores. Aunque para diagnosticar problemas en redes de comunicación, las capas inferiores son irrelevantes para la definición del protocolo HTTP .

Cliente: el agente del usuario

El agente del usuario, es cualquier herramienta que actué en representación del usuario. Esta función es realizada en la mayor parte de los casos por un navegador Web. Hay excepciones, como el caso de programas específicamente usados por desarrolladores para desarrollar y depurar sus aplicaciones.

El navegador es **siempre** el que inicia una comunicación (petición), y el servidor nunca la

comienza (hay algunos mecanismos que permiten esto, pero no son muy habituales).

Para poder mostrar una página Web, el navegador envía una petición de documento HTML al servidor. Entonces procesa este documento, y envía más peticiones para solicitar scripts, hojas de estilo (CSS), y otros datos que necesite (normalmente vídeos y/o imágenes).

El navegador, une todos estos documentos y datos, y compone el resultado final: la página Web. Los scripts, los ejecuta también el navegador, y también pueden generar más peticiones de datos en el tiempo, y el navegador, gestionará y actualizará la página Web en consecuencia.

Una página Web, es un documento de hipertexto (HTTP), luego habrá partes del texto en la página que puedan ser enlaces (links) que pueden ser activados (normalmente al hacer click sobre ellos) para hacer una petición de una nueva página Web, permitiendo así dirigir su agente de usuario y navegar por la Web. El navegador, traduce esas direcciones en peticiones de HTTP, e interpretará y procesará las respuestas HTTP, para presentar al usuario la página Web que desea.

El servidor Web

Al otro lado del canal de comunicación, está el servidor, el cual *"sirve"* los datos que ha pedido el cliente. Un servidor conceptualmente es una única entidad, aunque puede estar formado por varios elementos, que se reparten la carga de peticiones, (load balancing), u otros programas, que gestionan otros odenadores (como cache, bases de datos, servidores de correo electrónico, ...), y que generan parte o todo el documento que ha sido pedido.

Un servidor no tiene que ser necesariamente un único equipo físico, aunque sí que varios servidores pueden estar funcionando en un único computador. En el estándar HTTP/1.1 y Host , pueden incluso compartir la misma dirección de IP.

Proxies

Entre el cliente y el servidor, además existen distintos dispositivos que gestionan los mensajes HTTP. Dada la arquitectura en capas de la Web, la mayoría de estos dispositivos solamente gestionan estos mensajes en los niveles de protocolo inferiores: capa de transporte, capa de red o capa física, siendo así transparentes para la capa de comunicaciones de aplicación del HTTP, además esto aumenta el rendimiento de la comunicación. Aquellos dispositivos, que sí operan procesando la capa de aplicación son conocidos como proxies. Estos pueden ser transparentes, o no (modificando las peticiones que pasan por ellos), y realizan varias funciones:

- caching (la caché puede ser pública o privada, como la caché de un navegador)
- filtrado (como un anti-virus, control parental, ...)
- balanceo de carga de peticiones (para permitir a varios servidores responder a la carga total de peticiones que reciben)
- autentificación (para el control al acceso de recursos y datos)
- registro de eventos (para tener un histórico de los eventos que se producen)

Características clave del protocolo HTTP

HTTP es sencillo

Incluso con el incremento de complejidad, que se produjo en el desarrollo de la versión del protocolo HTTP/2, en la que se encapsularon los mensajes, HTTP está pensado y desarrollado para ser leído y fácilmente interpretado por las personas, haciendo de esta manera más fácil la depuración de errores, y reduciendo la curva de aprendizaje para las personan que empieza a trabajar con él.

HTTP es extensible

Presentadas en la versión HTTP/1.0, las cabeceras de HTTP, han hecho que este protocolo sea fácil de ampliar y de experimentar con él. Funcionalidades nuevas pueden desarrollarse, sin más que un cliente y su servidor, comprendan la misma semántica sobre las cabeceras de HTTP.

HTTP es un protocolo con sesiones, pero sin estados

HTTP es un protocolo sin estado, es decir: **no guarda ningún dato entre dos peticiones en la misma sesión.** Esto plante la problemática, en caso de que los usuarios requieran interactuar con determinadas páginas Web de forma ordenada y coherente, por ejemplo, para el uso de "cestas de la compra" en páginas que utilizan en comercio electrónico. Pero, mientras HTTP ciertamente es un protocolo sin estado, el uso de HTTP cookies, si permite guardar datos con respecto a la sesión de comunicación. Usando la capacidad de ampliación del protocolo HTTP, las cookies permiten crear un contexto común para cada sesión de comunicación.

HTTP y conexiones

Una conexión se gestiona al nivel de la capa de trasporte, y por tanto queda fuera del alcance del protocolo HTTP. Aún con este factor, HTTP no necesita que el protocolo que lo sustenta mantenga una conexión continua entre los participantes en la comunicación,

solamente necesita que sea un protocolo fiable o que no pierda mensajes (como mínimo, en todo caso, un protocolo que sea capaz de detectar que se ha pedido un mensaje y reporte un error). De los dos protocolos más comunes en Internet, TCP es fiable, mientras que UDP, no lo es. Por lo tanto, HTTP se apoya en el uso del protocolo TCP, que está orientado a conexión, aunque una conexión continua no es necesaria siempre.

En la versión del protocolo HTTP/1.0, habría una conexión TCP por cada petición/respuesta intercambiada, presentando esto dos grandes inconvenientes: abrir y crear una conexión requiere varias rondas de mensajes y por lo tanto resultaba lento. Esto sería más eficiente si se mandaran varios mensajes.

Para atenuar estos inconvenientes, la versión del protocolo HTTP/1.1 presentó el 'pipelining' y las conexiones persistentes: el protocolo TCP que lo transmitía en la capa inferior se podía controlar parcialmente, mediante la cabecera 'Connection'. La versión del protocolo HTTP/2 fue más allá y usa multiplexación de mensajes sobre una única conexión, siendo así una comunicación más eficiente.

Todavía hoy se sigue investigando y desarrollando para conseguir un protocolo de transporte más conveniente para el HTTP. Por ejemplo, Google está experimentado con QUIC, que se apoya en el protocolo UDP y presenta mejoras en la fiabilidad y eficiencia de la comunicación.

¿Qué se puede controlar con HTTP? (No lo veremos, pertenece más a sistemas)

La característica del protocolo HTTP de ser ampliable, ha permitido que durante su desarrollo se hayan implementado más funciones de control y funcionalidad sobre la Web: caché o métodos de identificación o autentificación fueron temas que se abordaron pronto en su historia. Al contrario, la relajación de la restricción de origen solo se ha abordado en los años de la década de 2010.

Se presenta a continuación una lista con los elementos que se pueden controlar con el protocolo HTTP:

- *Cache*
  El cómo se almacenan los documentos en la caché, puede ser especificado por HTTP. El servidor puede indicar a los proxies y clientes, que quiere almacenar y durante cuánto tiempo. Aunque el cliente, también puede indicar a los proxies de caché intermedios que ignoren el documento almacenado.
- *Flexibilidad del requisito de origen*
  Para prevenir invasiones de la privacidad de los usuarios, los navegadores Web, solamente permiten a páginas del mismo origen, compartir la información o datos. Esto es una complicación para el servidor, asi que mediante cabeceras HTTP, se puede flexibilizar o relajar esta división entre cliente y servidor

- *Autentificación*
  Hay páginas Web, que pueden estar protegidas, de manera que solo los usuarios autorizados puedan acceder. HTTP provee de servicios básicos de autentificación, por ejemplo mediante el uso de cabeceras como: WWW-Authenticate, o estableciendo una sesión especifica mediante el uso de HTTP cookies.
- *Proxies y tunneling*
  Servidores y/o clientes pueden estar en intranets y esconder así su verdadera dirección IP a otros. Las peticiones HTTP utilizan los proxies para acceder a ellos. Pero no todos los proxies son HTTP proxies. El protocolo SOCKS, por ejemplo, opera a un nivel más bajo. Otros protocolos, como el FTP, pueden ser servidos mediante estos proxies.
- *Sesiones*
  El uso de HTTP cookies permite relacionar peticiones con el estado del servidor. Esto define las sesiones, a pesar de que por definición el protocolo HTTP es un protocolo sin estado. Esto es muy útil no sólo para aplicaciones de comercio electrónico, sino también para cualquier sitio que permita configuración al usuario.

Flujo de HTTP

Cuando el cliente quiere comunicarse con el servidor, tanto si es directamente con él, o a través de un proxy intermedio, realiza los siguientes pasos:

1. Abre una conexión TCP: la conexión TCP se usará para hacer una petición, o varias, y recibir la respuesta. El cliente puede abrir una conexión nueva, reusar una existente, o abrir varias a la vez hacia el servidor.
2. Hacer una petición HTTP: Los mensajes HTTP (previos a HTTP/2) son legibles en texto plano. A partir de la versión del protocolo HTTP/2, los mensajes se encapsulan en franjas, haciendo que no sean directamente interpretables, aunque el principio de operación es el mismo.

GET / HTTP/1.1

Host: developer.mozilla.org

Accept-Language: fr

3. Leer la respuesta enviada por el servidor:

HTTP/1.1 200 OK

Date: Sat, 09 Oct 2010 14:28:02 GMT

Server: Apache

Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT

ETag: "51142bc1-7449-479b075b2891b"

Accept-Ranges: bytes

Content-Length: 29769

Content-Type: text/html

<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)

4. Cierre o reutilización de la conexión para futuras peticiones.

Si está activado el HTTP *pipelining*, varias peticiones pueden enviarse sin tener que esperar que la primera respuesta haya sido satisfecha. Este procedimiento es difícil de implementar en las redes de computadores actuales, donde se mezclan software antiguos y modernos. Así que el HTTP *pipelining* ha sido substituido en HTTP/2 por el multiplexado de varias peticiones en una sola trama.

Mensajes HTTP

En las versiones del protocolo HTTP/1.1 y anteriores los mensajes eran de formato texto y eran totalmente comprensibles directamente por una persona. En HTTP/2, los mensajes están estructurados en un nuevo formato binario y las tramas permiten la compresión de las cabeceras y su multiplexación. Así pues, incluso si solamente parte del mensaje original en HTTP se envía en este formato, la semántica de cada mensaje es la misma y el cliente puede formar el mensaje original en HTTP/1.1. Luego, es posible interpretar los mensajes HTTP/2 en el formato de HTTP/1.1.

Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato. Nosotros solo vamos a usar los métodos GET y POST.

Formato de las peticiones, el método GET y manejo de URLs

Toda petición HTTP (realizada, por ejemplo, al escribir una dirección web en la barra de direcciones de un navegador y pulsar enter) internamente no es más que un conjunto de líneas de texto.

En las peticiones GET, los cuerpos suelen estar vacíos y solo “piden” el recurso especificado.

La primera línea siempre define el tipo de petición y tiene el formato siguiente:

<MÉTODO> <URI> <VERSIÓN>.

Las siguientes líneas (que podemos dividir por su significado y contenido entre "cabecera de la petición" (puede haber una o varias cabeceras) y "cuerpo de la petición") aportan información adicional. Todas estas líneas se distinguen entre sí por su final, compuesto de los caracteres "\r\n" (retorno de carro más salto de línea).

El campo <VERSIÓN> simplemente indica la versión del protocolo utilizada para realizar la petición. El campo <MÉTODO> representa la acción que el cliente pretende ejecutar sobre un determinado recurso indicado mediante el campo <URI>. Por ejemplo, lo más habitual -con diferencia- es que el cliente quiera obtener una página web; en ese caso, como <MÉTODO> se usará el verbo GET –en mayúsculas- y como <URI> se indicará la dirección de la página web a conseguir. Existen varios métodos posibles más que permiten al cliente interactuar con el servidor de otras formas, como PUT, DELETE...

Tal como hemos dicho, el campo <URI> sirve, en el caso de usar el método GET, para indicar al servidor el recurso solicitado (el cual puede ser una página web, una fotografía...y en general, cualquier tipo de fichero). El formato de este campo (cuyo nombre viene de "Uniform Resource Identificator") está definido en el RFC 3986 (http://www.ietf.org/rfc/rfc3986.txt) pero, en la mayoría de ocasiones, utilizaremos un caso particular de URI llamado URL (de "Uniform Resource Locator") concretado en el RFC 1738 (http://www.ietf.org/rfc/rfc1738.txt) y que, en general, tiene este aspecto...:

protocolo://maquinaremota:puerto/ruta/recurso?c1=v1&c2=v2&c3=v3

- protocolo en nuestro caso siempre será la cadena "http" (aunque podría ser "https" si la conexión se estableciese sobre un canal cifrado).
- "maquinaremota" es el nombre DNS o dirección IP del servidor web.
- "puerto" es el número donde estará escuchando ese servidor (si dicho servidor usara

el puerto número 80 no hará falta indicarlo).

- "/ruta/recurso" es la ruta dentro de la jerarquía de carpetas del servidor web donde se aloja el recurso en cuestión y por último, si ese recurso fuera de tipo dinámico (es decir, si fuera un programa ejecutable que cada vez pudiera retornar un resultado variable, como es por ejemplo una página PHP) y para solicitarlo/ejecutarlo se usara el método GET, se le podrían pasar parámetros de entrada mediante una cadena final con siguiente formato (fíjate en la sintaxis: la primera pareja clave/valor va precedida de "?" y después todas las demás están separadas entre sí por "&"; a esta cadena de parejas clave/valor enviada al servidor para ser procesada se le llama "querystring").

?clave=valor&otraclave=otrovalor&otramás=suvalor&yotra=suvalor...

Por ejemplo, si escribimos en la barra de direcciones de un navegador una URI real tal como http://www.google.com/search?q=guapo y pulsamos enter, éste generará automáticamente una petición HTTP de tipo GET (este método es el utilizado por defecto por los navegadores si no se indica lo contrario) para acceder a un recurso llamado "search" ubicado en el servidor "www.google.com" (en otras palabras, al buscador de Google), al cual le estaremos indicando una pareja clave-valor (concretamente, la pareja "q=guapo"); con esta querystring, el buscador de Google estará recibiendo la información necesaria para realizar una búsqueda (clave "q") usando la palabra "guapo". De hecho, podemos comprobar fácilmente cómo, si accedemos a la página del buscador de Google a través de un navegador normal, dependiendo de lo que escribimos en el cuadro de búsqueda, así se modifica la cola "search?q=" de la dirección visible en la barra de direcciones.

Es importante tener en cuenta que un ordenador que actúe como servidor web normalmente tiene sus recursos compartidos (páginas web, imágenes, documentos, etc.) alojados dentro de una carpeta "raíz" determinada (establecida en la configuración del programa Apache/Nginx/IIS/etc.), cuya ruta real dentro de su sistema operativo no es conocida por el cliente porque este solamente "ve" a partir de esa carpeta en adelante.

Por ejemplo, si esa carpeta tuviera de ruta real "/var/www/html/misitioweb" y ahí dentro hubiera una subcarpeta "imagenes" con todas las imágenes de nuestro sitio web, para solicitar una imagen llamada "mifoto.png", en el navegador deberíamos escribir

http://www.miservidor.com/imagenes/mifoto.png y no http://www.miservidor.com/var/www/html/misitioweb/imagenesmifoto.png.

Hay que hacer notar también, la diferencia de formato existente entre las URL escritas en la barra de un navegador (las cuales tienen el aspecto ya conocido de protocolo://maquinaremota...) y las URL que aparecen formando parte de la primera línea interna de una petición HTTP, en las que se omite el protocolo, el nombre/dirección del servidor y su puerto de escucha (quedando, por tanto, con un aspecto similar a

"/ruta/recurso?c1=v1&c2=v2&c3=v3").

Esta diferencia es debida a que los valores omitidos en la URL de la petición HTTP ya son especificados al establecer la conexión entre cliente y servidor (proceso que es previo al envío de la petición propiamente dicha) y, por tanto, volverlos a indicar sería redundante.

Por ejemplo, la primera línea de una petición HTTP de tipo GET que solicitara (a un determinado servidor ya contactado) una página llamada "index.html" ubicada directamente dentro de la carpeta "raíz", debería tener un aspecto como:

人人生來自由，
在尊嚴和權利上一律平等。
他們有理性和良心，
請以手足關係的精神相對待。

GET /index.html HTTP/1.1.

Así pues, y solo para acabar de aclarar este aspecto:

¿cuál debería ser la petición GET necesaria para obtener, por ejemplo, la página http://arduino.cc/en/Reference/HomePage?

Respuesta: GET /en/Reference/HomePage HTTP/1.1

Finalmente, debemos saber que no todos los caracteres están permitidos en una URL: solamente podemos utilizar las letras del alfabeto inglés minúsculas y mayúsculas, los dígitos del 0 al 9, los caracteres -_ . ! * ' () y ciertos caracteres con significado especial dentro de las URLs, los cuales son:

Espacio en blanco, ",#,$,%,&,+,´,/,:,;,<,=,>,?,@,[,\,],{,|,} y ~.

Si deseáramos escribir alguno de estos últimos sin mantener su significado especial (es decir, si quisiéramos, por ejemplo, que "?" dejara de marcar el inicio de la "querystring" para pasar a ser un simple interrogante) deberíamos "codificarlos" según la tabla mostrada a continuación:

Carácter	Codificación
	%20
"	%22
#	%23
$	%24
%	%25
&	%26
+	%2B
,	%2C
/	%2F
:	%3A
;	%3B
<	%3C
=	%3D
>	%3E
?	%3F
@	%40
[	%5B
\	%5C
]	%5D
{	%7B
|	%7C
}	%7D
~	%7E

Así pues, si una página web tuviera por ejemplo un espacio en blanco en su nombre (pongamos que se llama "datos clientes.html"), una URL válida incluiría dicho nombre transformado así: "datos%20clientes.html".

Más ejemplos:

GET... images/logo.png (sin parámetros)

GET... /idex.php?page=main&lang=es (con dos parámetros, page con valor main y lang con valor es)

Las peticiones POST

Además del método GET, otro método ampliamente utilizado es el método POST. Este método está específicamente diseñado para enviar información desde el cliente (contenida en el cuerpo de la petición) hacia el recurso identificado por <URI> (ubicado en el servidor), para que éste la procese según lo tenga programado.

Un caso típico es el envío de datos desde un formulario web hacia una página PHP, ya que lo que hace es mandar “datos” a dicho servidor, lo hace en el cuerpo y puede “crear algo nuevo” o actualizar algo existente.

Pero hemos visto que el método GET también permite que un cliente pueda enviar datos al servidor mediante la creación de una querystring al final de la URL del recurso solicitado.

Entonces, ¿cuál es la diferencia entre ambos métodos a la hora de enviar datos al servidor? ¿Cuándo convendrá utilizar un método y cuándo otro?

Pues la diferencia principal está en el lugar dentro de la petición donde se encuentran los datos a enviar al servidor: con el método GET se envían, tal como hemos dicho, dentro de la propia URL solicitada (en forma de querystring y, por tanto, visibles para todo el mundo en la barra de direcciones del navegador) y con el método POST se envían dentro del cuerpo de la petición (y por tanto, permanecen algo más internos).

Los datos enviados mediante peticiones GET están escritos en un formato llamado "URL-encode"; esto significa que cumplen dos requisitos: siguen la estructura –ya vista- de una querystring y usan las reglas de codificación de caracteres mostradas en la tabla anterior.

Los datos enviados mediante POST pueden estar escritos igualmente en el formato "URL-encode" (es decir, seguir la estructura de querystring y estar codificados convenientemente -eso sí, dentro del cuerpo de la petición-) pero también pueden tener otro formato más específico llamado "multipart", diseñado especialmente para la transferencia de datos binarios (útil, pues, para la transmisión de ficheros) u otros más recientes, como JSON.

El formato utilizado por defecto en todas las peticiones POST, de todas formas, es siempre el "URL-encode".

Cabeceras de las peticiones

Las peticiones HTTP no se componen solamente de una sola línea de tipo <MÉTODO> <URI> <VERSIÓN> sino que están formadas por más líneas, las cuales pueden formar parte o bien de la llamada "cabecera" de la petición o bien del "cuerpo" de la petición. La separación entre cabecera y cuerpo se realiza gracias a una línea en blanco (es decir, una línea que contiene solamente los caracteres "\r\n" y ninguno más) entre ambas secciones.

Las líneas de cuerpo solamente aparecen en peticiones de tipo POST porque son las que contienen los datos que el cliente envía al servidor (tal como ya se ha comentado).

Las líneas de la cabecera sirven para informar al servidor sobre detalles técnicos de la petición, permitiendo que éste pueda componer una respuesta más adecuada. Todas estas líneas tienen la forma Nombre: Valor (no se distinguen mayúsculas de minúsculas) y están definidas en el documento RFC 2616.

Todas ellas son opcionales excepto una: la línea "Host:xxxx" (donde "xxxx" representa en este caso el nombre DNS -o dirección IP- del servidor web al que el cliente quiere conectar). A continuación, se lista una (muy) breve selección de las líneas de cabeceras más comunes:

Host: Sirve para indicar, como hemos dicho, el nombre DNS (seguido de ":" y un no de puerto si éste fuera diferente de 80) del servidor HTTP al que se le envía la petición. Por ejemplo, Host: elpuig.xeill.net:4567 Esta cabecera puede parecer redundante, pero en casos donde un servidor HTTP con una única dirección IP tiene asociados varios nombres DNS (algo bastante habitual) esta cabecera permite localizar correctamente el recurso indicado en la URI. Es la única línea de cabecera obligatoria.

Accept: Sirve para indicar al servidor, separados por comas, qué tipos MIME (de los recursos solicitados) el cliente es capaz de aceptar. Los recursos cuyo tipo MIME no esté incluido en la lista especificada en esta línea de cabecera, serán rechazados por el cliente. Un ejemplo (que solamente acepta páginas HTML) podría ser este: Accept: text/html.

NOTA: Un tipo MIME es una cadena (cuyo formato está definido en el documento RFC 1341 (https://www.ietf.org/rfc/rfc1341.txt) que sirve para clasificar un recurso según su naturaleza: texto, imagen, audio... Cada tipo MIME consta de un tipo principal genérico

("text", "image", "audio"...) y un subtipo más concreto (adecuado al tipo principal) que indica el formato específico de dicho recurso.

La lista oficial de tipos MIME está en la web de la organización internacional IANA (aquí: http://www.iana.org/assignments/media-types/media-types.xhtml) pero algunos de los tipos MIME más habituales son: text/plain (texto plano genérico), text/html (código HTML), text/css (código CSS), application/javascript (código Javascript), application/json (datos de tipo JSON), application/x-www-urlencoded (datos en formato "URL-encode"), text/csv (datos en formato CSV), image/gif (imagen GIF), image/jpeg (imagen JPG), image/png (imagen PNG), application/pdf (documento PDF), application/gzip (datos comprimidos de tipo Gzip) o application/octet-stream (datos sin estructura reconocida), entre otros. Para indicar en conjunto todos los subtipos de un tipo dado, se puede usar el símbolo especial "*", así, por ejemplo: image/*. Igualmente, para indicar todos los tipos MIME posibles, se puede escribir */*.

Content-Type: Esta línea de cabecera solamente aparece en peticiones de tipo POST y sirve para informar al servidor sobre el tipo MIME de los datos enviados en el cuerpo de la petición (el cual suele ser "application/x-www-urlencoded" o "application/json").

Si están en formato "URL-encode" (que es lo predeterminado), el valor de esta línea de cabecera será, por tanto, Content-Type: application/x-www-form-urlencoded. Siempre viene acompañada de otra línea de cabecera llamada Content-Length, la cual sirve para informar al servidor del tamaño (en bytes) del cuerpo de la petición (es decir, de los datos transferidos); un ejemplo: Content-Length: 348

Date: Sirve para informar al servidor de la fecha y hora en la que la petición fue enviada. Su valor se ha de expresar (en las pocas ocasiones que pueda ser necesario indicar esta línea de cabecera) en un formato (definido en el documento RFC 1123 (http://www.ietf.org/rfc/rfc1123.txt) que tiene el siguiente aspecto general: "díadelasemana, díadelmes mes año hora: minuto: segundo GMT". Por ejemplo, Date: Sat, 6 Jun 2016 10:10:10 GMT.

Referer: Sirve para informar al servidor de la URI del recurso que fue solicitado justo antes del recurso actual. En la práctica, su valor suele ser la dirección de la página web que contenía el enlace que ha llevado a la petición vigente hacia la página actual. Por ejemplo, Referer: http://www.rebelion.org

User-Agent: Sirve para informar al servidor sobre qué programa concreto (y su versión) es el cliente que realiza la petición. Por ejemplo, User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:32.0) Firefox/32.0. Esta línea de cabecera nos puede venir bien para simular ser un determinado navegador y comprobar el comportamiento del servidor según este dato.

Códigos de las respuestas

La respuesta HTTP enviada por un servidor a un cliente (correspondiente a la petición hecha anteriormente por éste) también se compone internamente de varias líneas de texto terminadas con los caracteres "\r\n".

La primera de estas líneas siempre define el tipo de respuesta ofrecida y tiene el formato siguiente: <VERSIÓN (versión http)> <CÓDIGO (estado)> <FRASE (explicación)>.

Las siguientes líneas (que podemos dividir en cabecera de la respuesta y cuerpo de la respuesta) aportan, respectivamente, información sobre detalles más técnicos de la respuesta y el contenido propiamente dicho de ésta.

El campo <VERSIÓN> simplemente indica la versión del protocolo utilizada para realizar la respuesta, y siempre tendrá como valor la cadena "HTTP/1.1". El campo <CÓDIGO> es un número de tres dígitos que define qué tipo de respuesta se está proporcionando. El campo <FRASE> contiene una breve explicación textual del valor de <CÓDIGO>; el documento RFC 2616 da recomendaciones sobre estas frases, pero son opcionales y pueden cambiarse como se estime oportuno.

Los valores posibles del campo <CÓDIGO> se pueden clasificar en cinco categorías, dependiendo del resultado obtenido en el servidor tras el procesamiento de la petición. Estas categorías vienen identificadas por el dígito de más a la izquierda, sirviendo los otros dos para especificar más al detalle el tipo de respuesta concreta.

A continuación, presentamos algunos de los códigos de respuesta comunes usados para explicar lo sucedido:

1xx (Información): El servidor indica que la petición ha sido recibida y que procederá a procesarla. Se usa en casos muy específicos que no veremos.

2xx (Éxito): El servidor indica que la petición ha sido recibida y procesada con éxito. El código más típico de esta categoría es 200, obtenido cuando todo ha sido satisfactorio y el recurso solicitado es devuelto correctamente (si el método de la petición fue GET, se devuelve el recurso en sí; si fue POST, el resultado de la acción, etc.)

3xx (Redirección): El servidor indica que la petición no se ha completado y se deben realizar más acciones para conseguir finalizarla. Generalmente se usa como respuesta a peticiones GET para señalar un cambio de localización del recurso solicitado (es decir, una nueva URI de una página ya existente). Esta nueva localización se indica en una cabecera enviada por el servidor justo para ese propósito: Location. Si el cambio es permanente, se emplea el código 301 y para peticiones siguientes el cliente debe usar la ubicación proporcionada en Location; si el cambio es temporal existen otros códigos (302, 303, 307...) y las siguientes peticiones deben seguir haciéndose a la URI original.

Normalmente estos códigos son interpretados automáticamente por cualquier navegador actual, realizando la acción pertinente sin necesidad de notificarlo al usuario.

4xx (Error de cliente): El servidor indica que la petición proveniente del cliente está mal formada o contiene errores. El código devuelto más común en estos casos es el 404, que indica que el servidor no ha encontrado el recurso solicitado en la URI de la petición. Pero hay muchos más: el código 400 indica que la sintaxis de la petición HTTP es errónea; el código 401 indica que el usuario no ha presentado las credenciales adecuadas para poder estar autorizado a acceder a un recurso protegido (el proceso de autenticación y autorización de recursos está definido en un documento RFC específico, el 2617 (https://www.ietf.org/rfc/rfc2617.txt); el código 403 indica que el recurso está protegido sin posibilidad de presentar credenciales, etc.

5xx (Error de servidor): El servidor indica que, aunque la petición es correcta, ha fallado al procesarla. Los distintos motivos se señalan con un código concreto. Por ejemplo, 500 avisa de un error indeterminado o 501 indica que el servidor no soporta el método de la petición, entre otros.

Ejemplo:

HTTP/1.1 200 ok

Content-Type: text/xml; charset=utf-8

content-length: length

<html>

...

<html>

Cabeceras de las respuestas

Las respuestas HTTP no se componen solamente de una sola línea de tipo <VERSIÓN> <CÓDIGO> <FRASE> sino que, tal como ya se ha dicho, están formadas por más líneas, las cuales pueden formar parte o bien de la llamada "cabecera" de la respuesta o bien del "cuerpo" de la respuesta. La separación entre cabecera y cuerpo se realiza gracias a la existencia de una línea en blanco (es decir, una línea que contiene solamente los caracteres "\r\n" y ninguno más) entre ambas secciones.

Las líneas de cuerpo solamente aparecen en respuestas a peticiones de tipo GET y son el

contenido del recurso propiamente solicitado (es decir, lo que habitualmente suele ser el código de una página web). Este contenido es el que los navegadores obtienen, interpretan y, como resultado, muestran en pantalla. Así pues, aunque el cuerpo de las respuestas a peticiones GET no está ceñido a ningún formato en particular; en cada respuesta el servidor deberá indicar al cliente el tipo MIME concreto del contenido enviado (mediante la directiva Content-Type, que veremos enseguida) para que dicho cliente lo pueda interpretar convenientemente al recibirlo.

Las líneas de la cabecera sirven para informar al cliente sobre detalles más técnicos de la respuesta, permitiendo que éste pueda obtener una información más completa sobre el recurso en cuestión. Todas estas líneas tienen la forma Nombre: Valor (no se distingue entre mayúsculas y minúsculas) y también están definidas en el documento RFC 2616. A continuación, se lista una breve selección de las líneas de cabeceras más habituales en las respuestas de los servidores web:

Connection: Sirve para informar al cliente sobre si la conexión TCP creada por el servidor para enviar la respuesta HTTP continuará abierta o no. Si es que sí (valor keep-alive), esa conexión TCP podrá ser reutilizada para recibir más peticiones HTTP posteriores del mismo cliente; si no (valor close), será cerrada y para enviar una nueva respuesta HTTP el servidor tendrá que crear una nueva conexión.

Content-Type: Sirve, en respuestas a peticiones GET, para informar al cliente sobre cuál es el tipo MIME del contenido del cuerpo del recurso devuelto. También informa (pero sólo si el tipo MIME es textual, como text/plain o text/html) de su sistema de codificación (ASCII, UTF-8, etc.), para que el cliente pueda mostrar dicho contenido correctamente. Esto último se indica mediante la palabra especial "charset" tras un punto y coma, así: Content-Type: text/html; charset=UTF-8 (si el sistema de codificación no se indicara, el cliente empleará uno por defecto que dependerá de su configuración, el cual puede no coincidir con el del recurso). Salvo raras excepciones, UTF-8 es el sistema de codificación recomendado por ser el más versátil, óptimo y compatible entre clientes.

Content-Length: Sirve para informar al cliente del tamaño (en bytes) del contenido del cuerpo del recurso devuelto (una vez comprimido, si es el caso). Si el valor de la línea Connection es close, el valor de esta línea es fácilmente calculable y se puede indicar directamente, así: Content-Length: 348. No obstante, si Connection es keep-alive (por ejemplo, cuando se ofrece un fichero en "streaming"), el servidor no sabe de antemano este dato, por lo que esta línea de cabecera suele ser sustituida por la línea Transfer-Encoding: chunked, la cual indica al cliente que el recurso es enviado "a pedazos" progresivamente hasta enviar una marca final.

Date: Sirve para informar al cliente de la fecha y hora en la que la respuesta fue generada. Su valor se ha de expresar en un formato definido en el documento RFC 1123 (igual que ocurre con la línea de cabecera de cliente homónima).

Expires: Sirve para informar al cliente de la fecha y hora en la que la respuesta dada se considerará caducada. Su valor se ha de expresar en un formato definido en el documento RFC 1123.

Location: Sirve para indicar al cliente la nueva URI de un recurso cuando la URI solicitada

(correspondiente a ese recurso) ya no es válida. En general, gracias a esta información un navegador actual será capaz de redireccionar automáticamente su petición a la URI recién indicada. Por ejemplo, Location: http://www.fbi.gov

Server: Sirve para informar al cliente del nombre y sistema del servidor. Por ejemplo, Server: Apache/2.4 (Unix)

2. ARQUITECTURA DEL SOFTWARE

a) En los inicios de la programación, se podía decir que programar era un arte. Programación= arte

Pocos sabían

Se desarrollaba como tal, arte en programar, dependía de las aptitudes de cada programador.

b) Ahora:

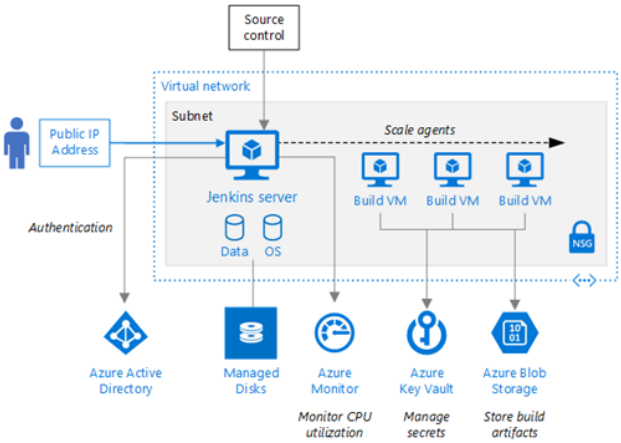
Guías, modelos, formas para resolver problemas típicos

Se usa la palabra arquitectura porque son como “planos de edificios”:

Estructura del edificio (Aplicación)

Funcionamiento de cada parte

Interacción entre partes



Más allá de algoritmos (código), el diseño y la estructura global de una aplicación se ha convertido en un nuevo problema por la envergadura y la cantidad de funcionalidad de las mismas.

c) Cada modelo para solucionar esto (hay muchos) tiene un número de "visiones", pero mínimo suele haber 3:

- Visión estática: Qué componentes necesitamos
- Visión funcional: Qué hace cada componente
- Visión dinámica: Cómo se comportan los componentes en el tiempo y cómo interactúan entre si.

Esto se suele expresar en muchos lenguajes, el más consensuado es el UML.

d) Arquitecturas más comunes:

- Descomposición modular
- Cliente servidor
- ...
- A 3 niveles o capas (una especialización de cliente-servidor) donde una capa solo tiene relación con la siguiente.

  Presentación (vista)

  Cálculo o lógica de negocio

  Persistencia o almacenamiento
- MVC (nos centraremos en esta)

3. MODELO VISTA CONTROLADOR

Se pueden ver enlaces en la plataforma.

3.1) Introducción

1. Es un patrón de diseño o arquitectura de sorftware, pero ¿qué es un patrón de diseño?

Solución a un problema DE DISEÑO con algunas características:

- Se debe haber comprobado su efectividad en problemas similares y en casos anteriores (Ej. Un baño de agua fría baja la fiebre).
- Reutilizable en problemas parecidos.

¿Qué pretenden los patrones de diseño?

Categorías de patrones.

De arquitectura

De diseño

Dialectos

(Se debe leer también sobre los antipatrones de diseño, por ejemplo, en wikipedia, en cuya documentación se dan argumentos para NO escoger determinados caminos de diseño, se habla del objeto todopoderoso, de incluir números concretos en el código sin explicación aparente (Números mágicos), etc.)

https://es.wikipedia.org/wiki/Antipatr%C3%B3n_de_dise%C3%B1o

3.2) Separa datos de la lógica de negocio

¿Qué es la lógica de negocio?

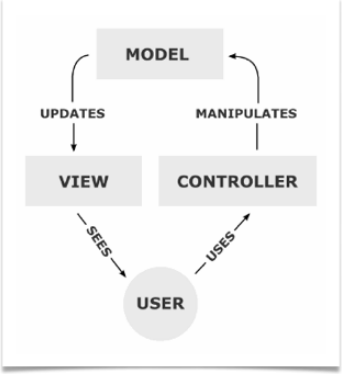
- Parte que codifica las **reglas** de negocio ¿Qué son estas reglas?
- Determinan qué información puede ser creada, modificada, mostrada, etc. en el mundo real.
- Corresponde al backend (parte que el usuario no ve de la aplicación).
- El usuario interactúa con la vista (interfaz) e introduce o recibe datos, pero el procesamiento de esos datos, son lo que gestiona la lógica de negocio.
- Funcionalidad ofrecida por el software.

Ejemplo de regla de negocio:

“Un cliente al que facturamos más de 10.000 al año es un cliente de tipo A”, "A los clientes de tipo A les aplicamos un descuento del 10% en pedidos superiores a 3.000."

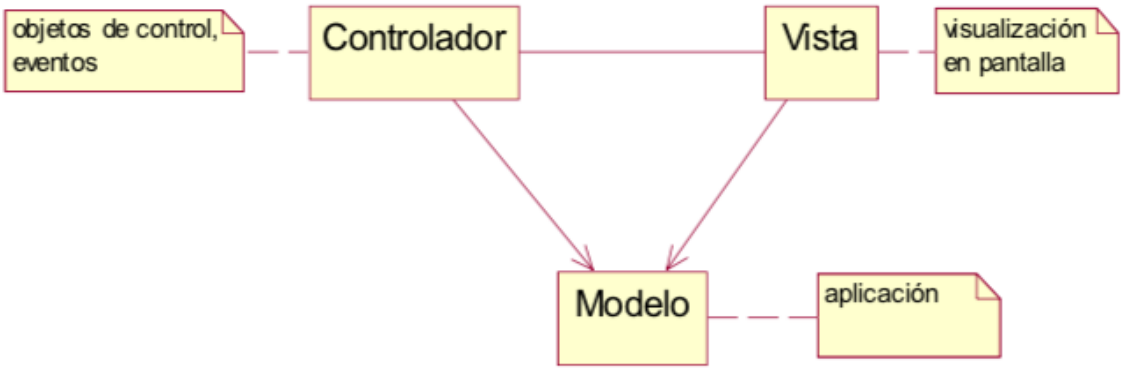
Más ejemplos de lógica de negocio serían agregar una línea de pedido a un pedido, calcular precios de productos mediante alguna regla, crear un acta de evaluación de alumnos donde se pueden ver clases enteras, información del tutor, estadísticas, etc.

Nota: En algunos manuales, se habla de la lógica de negocio en el modelo y el controlador es quien pone en contacto la vista con el modelo. Puede verse así en el sentido de todo lo que está relacionado con los datos. Nosotros la vemos en el controlador.



https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

Trata de distribuir responsabilidades.



4. DAO (DATA ACCESS OBJECT)

Es un componente de software que suministra una interfaz común entre la aplicación y uno o varios dispositivos de almacenamiento (base de datos, archivo, etc.)

Ventajas:

- Cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.
- Los DAO aislan la aplicación de la base de datos (la tecnología usada de persistencia como JDBC, JavaBeans, Hibernate, IBATIS...)

Desventajas:

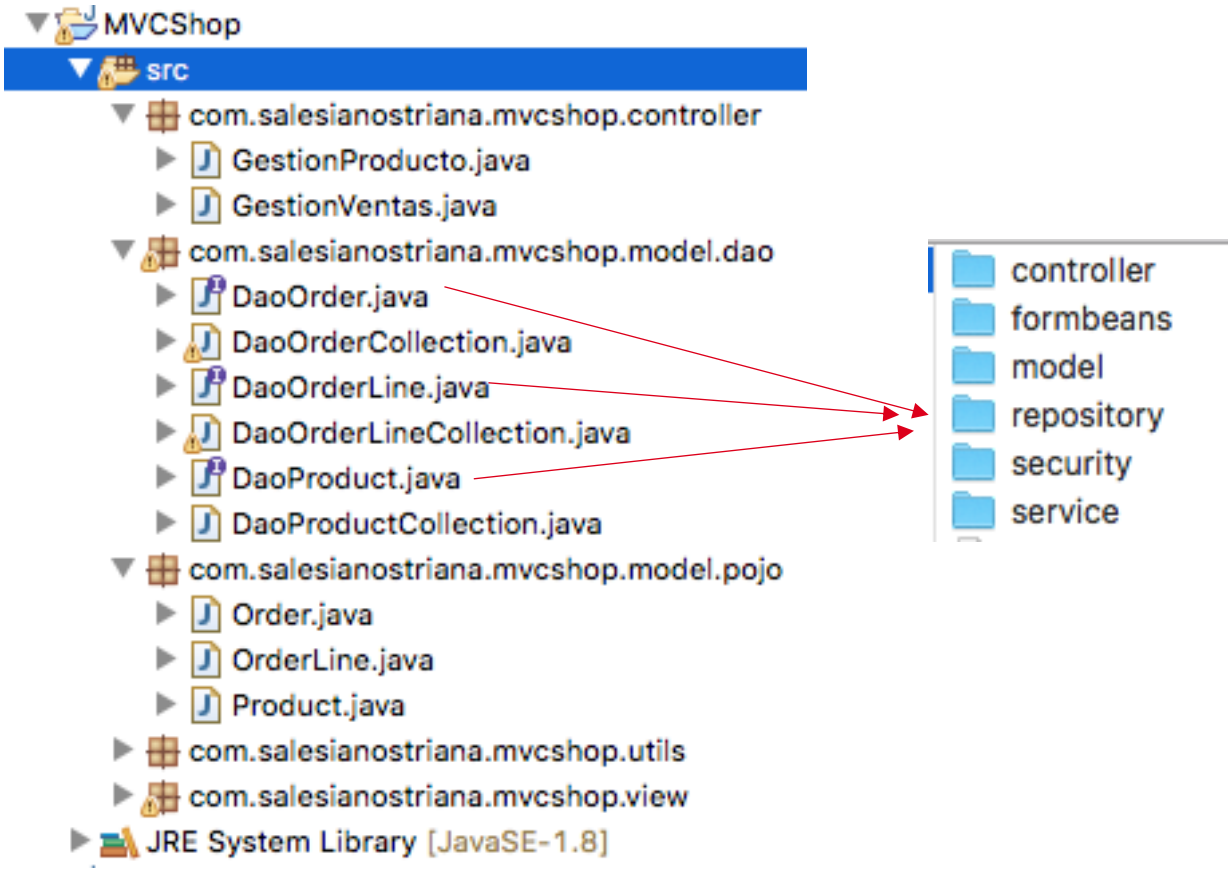
- La flexibilidad tiene un precio. Añadir otra capa incrementa la cantidad de código ejecutado en tiempo de ejecución.
- La configuración de las capas de persistencia requiere de mucho trabajo.

Por tanto, si necesitamos alto rendimiento, el uso de DAO no es la mejor opción.

5. EJEMPLO ESTRUCTURA DE UNA APLICACIÓN BASADA EN MVC Y DAO (MVCSHOP)

(La estructura de la izquierda, no será la estructura definitiva que usaremos en el proyecto final, ya que nuestros "DAO" no irán en el paquete model, sino en otros llamados repositorios, que serán los que ayudarán al modelo a acceder a la base de datos, figura de la derecha).

Aunque no sea nuestra estructura definitiva si es bueno que vayamos viendo estos pasos intermedios para tener una idea global sobre cómo se van aplicando los distintos patrones.



En GestionVentas, tendremos como atributos una línea de pedido y un pedido (del tipo interfaz DAO correspondiente). Por tanto, en “nuestra base de datos (que ahora mismo es una colección) insertaríamos una nueva venta a través del DAO que va agregando todas las líneas de pedido.

NOTA: Si quisiéramos guardar en otro sitio, por ejemplo, en un fichero CSV, un archivo XML, una base de datos JDBC, etc. Bastaría con tener las clases DAOProductCSV, DaoProductXML, DaoProductoHib, DaoProductJDBC... y estas últimas, deberían implementar la interfaz DaoProduct. Así, podríamos buscar de la misma manera independientemente de la tecnología que usemos para guardar los datos, sin “tocar” Product o DaoProduct.