# Fuente:

https://www.evaluandosoftware.com
https://codigofacilito.com
https://www.escuelapython.com
https://wiki.uqbar.org
https://anexsoft.com

## Arquitectura distribuida

La arquitectura web trabaja con dos nodos:

- el **cliente** tiene un programa ejecutable (application client, el *web browser* o navegador es el más común)
- y el **servidor** tiene otro programa ejecutable: será nuestro server.

Estos nodos son lógicos: pueden estar ubicados físicamente en la misma máquina, pero igualmente tendremos una separación de componentes en cliente y servidor.

El cliente hace pedidos a través de un puerto contra el servidor, el servidor responde. El flujo de mensajes siempre comienza en el cliente:

- **cliente** pide servicio (**request**)
- **servidor** responde (**response**)


## Algunas consecuencias

- Nuestra aplicación pasa a ser una **aplicación distribuida**: va a tener una parte corriendo en el servidor y otra parte corriendo en el cliente. Dependiendo de la arquitectura que elijamos
    - podemos tener la mayor parte de la lógica en el servidor y tener un cliente liviano (thin) o ZAC (Zero Administration Client). Entonces lo que le llega al cliente es sólo un documento HTML, y es fácil mantener la aplicación cuando tengo muchos clientes ubicados

    - o bien podemos poner gran parte de la lógica en el cliente y utilizar la parte server solamente para sincronizar la información entre sesiones de usuario
- de todas maneras, por más liviano que sea el cliente, los *browsers* no son uniformes, entonces si queremos que una aplicación ande en todos ellos muchas veces vamos a tener que manejar código específico para cada plataforma (browser, versión, sistema operativo y a veces hasta el hardware).
- como el cliente es el que dispara los pedidos, todas las interacciones entre el usuario y la aplicación deben ser iniciadas por el usuario, la aplicación no puede tomar la iniciativa. Ej: si tengo una lista de tareas pendientes, para que aparezca una nueva tarea hay que obligar al cliente a que dispare el refresh.

**Pedido/respuesta**

Antes de meternos más de lleno, nos preguntamos: la tecnología de objetos ¿es consistente con la metáfora “pedido-respuesta” (request/response)? Sí, en definitiva es la representación de lo que es un mensaje.

En la tecnología web siempre es el cliente el que pide y siempre el servidor el que responde.

**Cómo se implementa la comunicación**

El cliente dice: “necesito x”. Esto se traduce en una dirección de una página en particular, esa dirección recibe el nombre de **URL** (Uniforme Resource Locator, o forma de encontrar un recurso en el servidor):

**http://localhost:8080/html-css/index.html**

donde

- **http** es HyperText Transfer Protocol, el **protocolo de comunicación** por defecto que usan los navegadores
    - otros protocolos son: **https** (donde los datos viajan encriptados), **ftp**, etc.
- localhost es el **servidor web** hacia el que vamos a conectarnos
    - en este caso localhost es el web server que está en la PC local, que equivale a la dirección IP 127.0.0.1
    - el servidor web puede ser una dirección IP o un nombre que luego es convertido a una dirección IP a través de un DNS (Domain Name Server)

- 8080 es el puerto donde el servidor está “escuchando” pedidos
- y finalmente la página que queremos cargar, que recibe el nombre de **recurso**

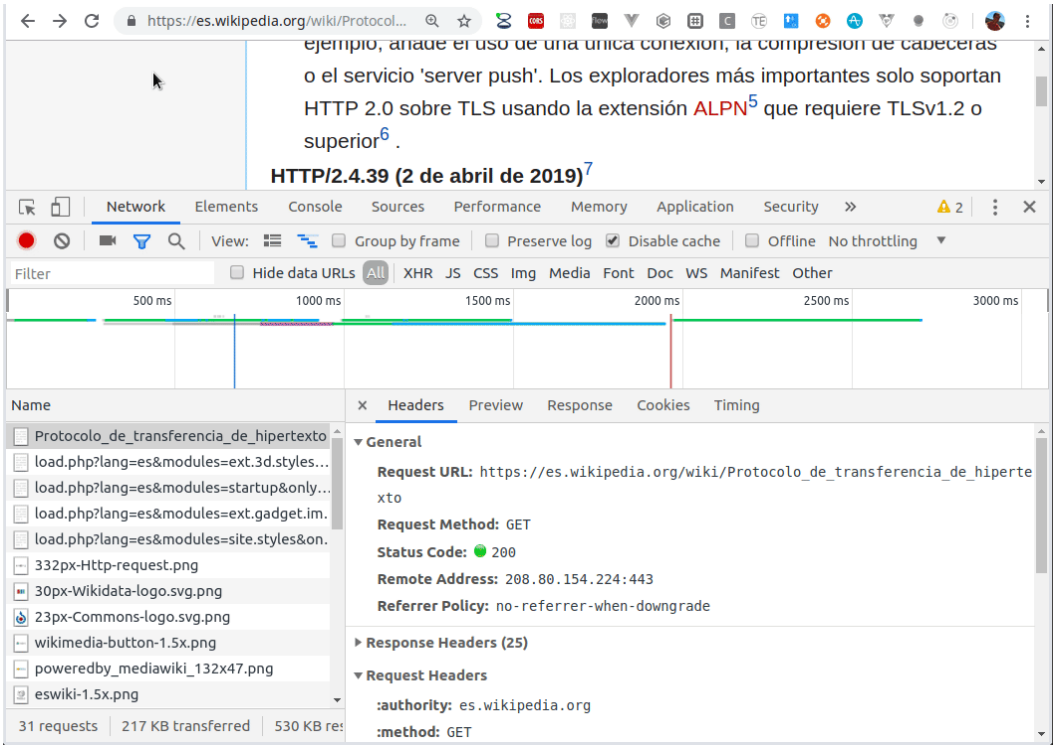
La forma en que publicaremos las páginas como **rutas** depende de la tecnología en la que trabajemos, lo importante es entender que una página html es accesible para un usuario con una ruta única llamada URL.

## HTTP

Http es un protocolo **no orientado a conexión** que define la forma de comunicación entre el cliente y el servidor.

No-orientado a conexión significa que no guarda ninguna información sobre conexiones anteriores, por lo que no tenemos el concepto de sesión de usuario, es un protocolo sin estado (*stateless protocol*). Esto tiene varias implicancias, la más fuerte es que requiere que la aplicación mantenga la información necesaria para mantener una sesión (por ejemplo, sabiendo qué usuario es el que está haciendo una operación).

Un mensaje http tiene formato de texto, por lo que es legible al usuario y fácilmente depurable, como vemos en el siguiente video:




Abrimos en un navegador las herramientas de desarrollo (por lo general es la tecla F12), y en la solapa *Network* podemos inspeccionar las distintas respuestas que procesa el navegador, con el pedido http original que hace un mensaje de tipo GET.

**Tipos de mensaje**

Un cliente puede enviar un pedido al servidor utilizando diferentes métodos

- **GET**: asociada a una operación de lectura, sin ningún otro efecto
- **HEAD**: es exactamente igual al pedido vía GET pero enviando únicamente el resultado de la operación en un header, sin el contenido o *body*
- **POST**: se suele asociar a una operación que tiene efecto colateral, no repetible
- **PUT**: está pensado para agregar información o modificar una entidad existente
- **DELETE**: se asocia con la posibilidad de eliminar un recurso existente
- **OPTIONS**: permite ver todos los métodos que soporta un determinado servidor web
- **TRACE**: permite hacer el seguimiento y depuración de un mensaje http (se agrega información de debug)
- **CONNECT**: equivalente a un ping, permite saber si se tiene acceso a un host

**Envío mediante GET method**

Aquí los parámetros viajan dentro de la URL como par clave=valor:

http://www.appdomain.com/users?size=20&page=5

- ? delimita el primer parámetro
- & delimita los siguientes parámetros

La ventaja de utilizar este método es que dado que http es un protocolo no orientado a conexión, podemos reconstruir todo el estado que necesita la página a partir de sus parámetros (es fácil navegar hacia atrás o adelante). Por otra parte es el método sugerido para operaciones sin efecto, que recuperan datos de un recurso.

Por otra parte, no es conveniente para pasar información sensible (como password o ciertos identificadores), algunos navegadores imponen un límite máximo de caracteres para estos pedidos y necesita codificar los caracteres especiales (p. ej. el espacio a %20) dado que el request solamente trabaja con el conjunto de caracteres ASCII.

**Envío mediante POST method**

Los parámetros viajan en el BODY del mensaje HTML, no se ven en la URL del browser. Aquí no hay restricciones de tamaño para pasaje de información y tampoco se visualizan los parámetros en la URL del browser.

**GET vs. POST**

La recomendación W3C (World Wide Web Consortium) dice que deberíamos usar

- **GET** cuando sepamos que se trata de consultas que no van a tener efecto colateral (no habrá modificación en el estado del sistema)
- **POST** cuando sepamos que el procesamiento de la página causará una alteración del estado del sistema (al registrar el alquiler de una película, al modificar los datos de un socio o al eliminar un producto de la venta). Otros métodos posibles que veremos son PUT y PATCH, para modificaciones y alteraciones parciales, respectivamente.

El camino de un pedido http

---

- el browser se conecta con el servidor a partir del dominio o IP (localhost = 127.0.0.1) y puerto
- se envía la petición al servidor en base a dirección, método, parámetros, etc.
- el servidor responde a ese pedido: esa respuesta es una nueva página con un código de estado HTTP:
    - 200 : OK
    - 401 : Unauthorized
    - 403 : Forbidden
    - 404 : Not Found
    - 405 : Method not allowed
    - 500 : Internal Server Error

El lector puede buscar la lista de códigos de error HTTP (las especificaciones RFC 2616 y RFC 4918) y formas de resolverlos.

- la aplicación cliente o *user agent* se desconecta del servidor una vez procesada la respuesta

## Consecuencias del mensaje http para las aplicaciones web

La página es la mínima unidad de información entre cliente y servidor, lo que implica:

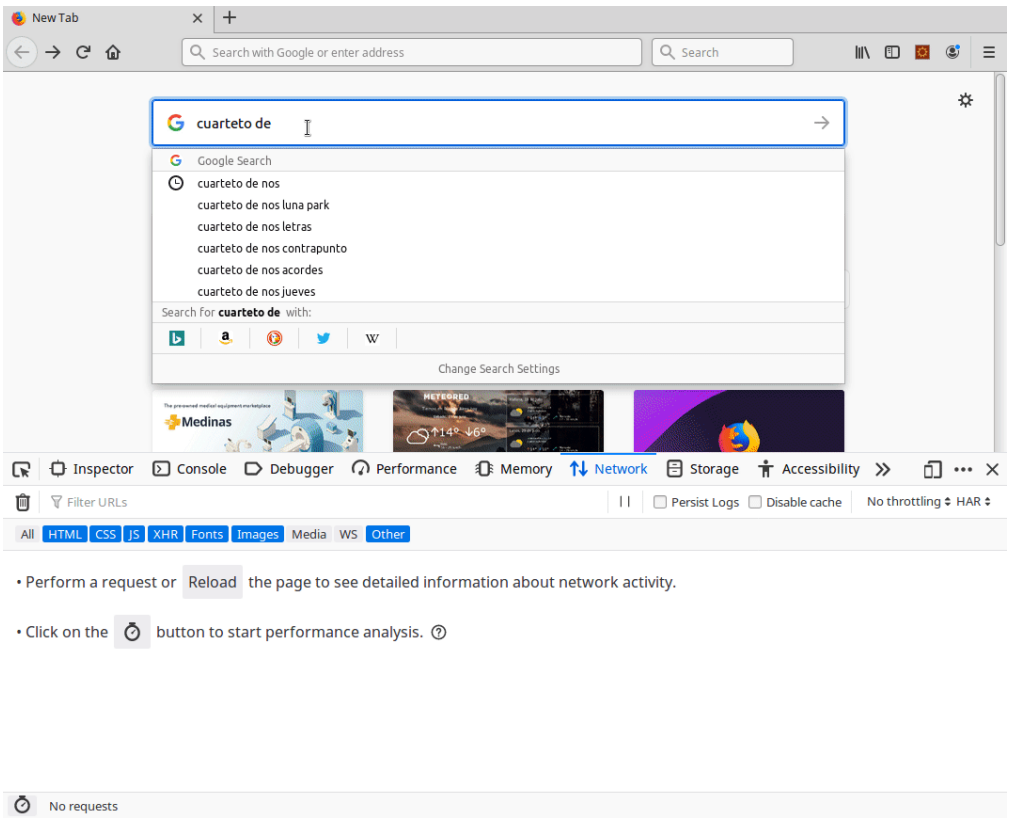
- **problemas en la performance**: no siempre debería refrescar toda la página si sólo necesito actualizar parcialmente la información de dicha página
- **problemas en el diseño**: tengo dificultades para poder particionar una pantalla en componentes visuales
- **problemas de usabilidad**: para que la página sea dinámica necesitamos forzar una comunicación con el servidor

**String oriented programming**: la comunicación entre cliente y servidor involucra solo texto, necesitamos adaptar fechas, números, booleanos y también los objetos de negocio (socios de un videoclub, alumnos, materias, vehículos de una flota, etc.) así como las colecciones.

## Procesamiento de la respuesta en el cliente

El servidor contesta con un string que tiene

- un ***header*** donde indica el resultado del pedido
- un **contenido**, que forma parte del body, que puede ser HTML, json o cualquier otro formato que el cliente entienda



Arriba vemos la respuesta del navegador al buscar “Cuarteto de Nos”.

**REST**

"Representational State Transfer" o traducido a "Transferencia de presentación de estado" es lo que se domina a REST. ¿**Y eso es**?, una técnica de arquitectura de software usada para construir APIs que permitan comunicar a nuestro servidor con sus clientes usando el protocolo HTTP mediante URIs lo suficientemente inteligentes para poder satisfacer la necesidad del cliente.

- REST es **STATELESS**, es decir que cada petición que reciba nuestra API **debe perecer**. Por ejemplo, **no podemos RECORDAR** un usuario logeado en el API usando una sesión, esto es un PECADO ya que agotaría la memoria RAM de nuestro servidor (10 mil usuarios conectados a nuestra API). Lo que correcto es pasar un TOKEN para cada petición realizada al API, y el API deberá validar si esta es correcta o no (por ahora no vamos hablar de técnicas para generar el TOKEN, pero lo más común es usar una COOKIE).
- Se implementan **RECURSOS** para generar comunicación, es decir crea URIs únicas que permiten al cliente entender y utilizar lo que está exponiendo. Por ejemplo:
    - api.anexsoft.com/users
    - api.anexsoft.com/users/1405
- Cada petición realizada a nuestra API responde a un verbo, y dicho verbo a una operación en común. Mediante los métodos HTTP hacemos las peticiones, lo común es GET y POST, PUT y DELETE.
    - **POST (create):** cuando mandamos información para insertar por ejemplo un registro en la base de datos. La información es enviada en el cuerpo de la petición, es decir que los datos no son visibles al usuario.
      api.anexsoft.com/users

- **GET (read)**: es usado para modo lectura, por ejemplo: cuando queremos listar a todos los usuarios de nuestra base de datos. Los parámetros son enviados por la URL.
  api.anexsoft.com/users
- **PUT (update):** cuando queremos actualizar un registro. Actualizar la información de un usuario X.
  api.anexsoft.com/users
- **DELETE (delete):** cuando queremos eliminar un registro. Borrar un usuario X de nuestra base de datos.
  api.anexsoft.com/users

Con esto hemos mencionado algunas caracteristicas básicas de lo que es REST, la cual podríamos decir que es un estándar para crear una REST API o RESTFul.

**¿QUÉ ES RESTFUL?**

Es un servicio que disponemos al público usando REST. REST es el concepto, RESTFul es la implementación y al crear un RESTFul creamos una API, la cual una API es un conjunto de funciones o procedimientos para que sea utilizado por otro software.

**Ej:** la API de Google Maps, Youtube, Facebook, etc ..

**¿BAJO QUÉ CIRTUSTANCIAS DEBERÍAMOS IMPLEMENTAR UN SERVICIO RESTFUL?**

Se dice que hoy en día debemos pensar orientado al servicio. Yo recomendaría trabajar directamente con una REST API bajo estas situaciones.

- Si vamos a trabajar una aplicación del tipo SPA, si o si debemos crear una REST API. Un framework recomendado puede ser AngularJS, EmberJS, backbone entre otros.
- Si es una Web comercial y vas a exponer recursos al público, es muy recomendable crear una REST API.
- Si tu sistema no solo va a ser accedido desde una PC, tambíen deberíamos pensar en crear una REST API, ya que permite crear escalabilidad.

En resumen, crear una REST API es una manera muy profesional de desacoplar tu sistema de la capa de persistencia.

**PUNTOS A FAVOR**

- **Separación entre el cliente y el servidor**
  Nuestros proyectos se vuelven autónomos, por lo tanto no nos interesa si dicha tecnología es compatible con la otra ya que usaremos como un medio de comunicación JSON.
- **No importa la tecnología**
  Si eres PHP, .NET, Java, Ruby, etc da igual, al final solo necesitas saber como consumir/responder al servicio.
- **Escalabilidad, Flexibilidad**
  Realiza los cambios que quieras dentro de tu Api, lo que interesa es que se respete el mismo mensaje o respuesta que le brindas al cliente para mantener la misma lógica.
- **¿Mejora de recursos consumidos por el servidor?**
    - REST no debe usar sesiones, por lo tanto disponemos de más memoria RAM.
    - Lo correcto es trabajar con formatos estandarizados como JSON no haremos uso de HTML para responder al cliente. En este caso ganamos velocidad.

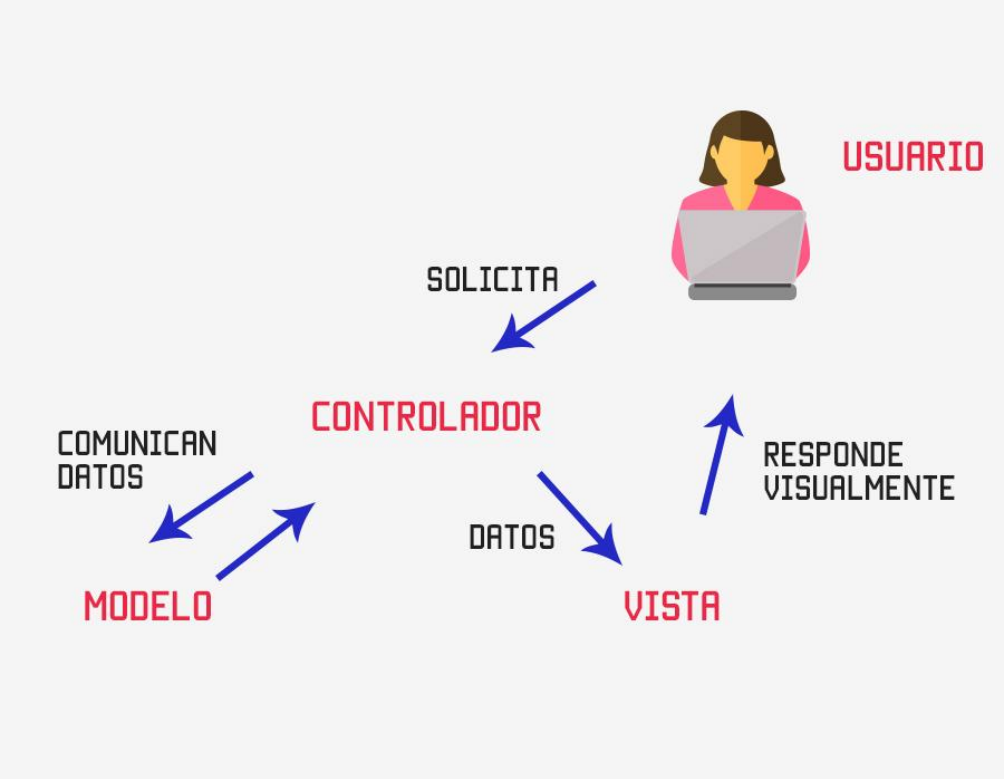
**PUNTOS EN CONTRA**

- Mayor tiempo en desarrollo debido a que hay que plantear y estandarizar las respuestas de nuestra API para que se tornen amigables para quien la consume.
- Curva de aprendizaje incrementada, ya que como muchos estan acostumbrados a trabajar enviado HTML por las peticiones AJAX, podría resultar un poco

tedioso trabajar con JSON puro, entre ellas, entender el uso del framework que implemente la API o tener claro en el concepto si queremos desarrollar un framework desde cero.

**MVC**

El MVC o Modelo-Vista-Controlador es un patrón de arquitectura de software que, utilizando 3 componentes (Vistas, Models y Controladores) separa la lógica de la aplicación de la lógica de la vista en una aplicación. Es una arquitectura importante puesto que se utiliza tanto en componentes gráficos básicos hasta sistemas empresariales; la mayoría de los frameworks modernos utilizan MVC (o alguna adaptación del MVC) para la arquitectura, entre ellos podemos mencionar a Ruby on Rails, Django, AngularJS y muchos otros más. En este pequeño artículo intentamos introducirte a los conceptos del MVC.




**UNA ANALOGÍA**

Una que me gusta mucho es la de la televisión. En tu televisión puedes ver distintos canales distribuidos por tu proveedor de cable o televisión (que representa al modelo), todos los canales que puedes **ver** son la vista, y tú cambiando de canal, **controlando qué ves** representas al controlador.

## LA EXPLICACIÓN

Los puntos anteriores so sólo para proveer background, y que ojalá puedas utilizar las referencias ahora que vamos a explicar qué es.

Antes que nada, me gustaría mencionar por qué se utiliza el **MVC**, la razón es que nos permite separar los componentes de nuestra aplicación dependiendo de la responsabilidad que tienen, esto significa que cuando hacemos un cambio en alguna parte de nuestro código, esto no afecte otra parte del mismo. Por ejemplo, si modificamos nuestra Base de Datos, sólo deberíamos modificar el modelo que es **quién se encarga de los datos** y el resto de la aplicación debería permanecer intacta. Esto respeta el principio de la responsabilidad única. Es decir, una parte de tu código no debe de saber qué es lo que hace toda la aplicación, sólo debe de tener una responsabilidad.

En web, el MVC funcionaría así. Cuando el usuario manda una petición al navegador, digamos quiere ver el curso de AngularJS, el controlador responde a la solicitud, porque él es el que controla la lógica de la app, una vez que el controlador nota que el usuario solicitó el curso de Angular, le pide al modelo la información del curso.

El modelo, que se encarga de los datos de la app, consulta la base de datos y digamos, obtiene todos los vídeos del curso de AngularJS, la información del curso y el título, el modelo responde al controlador con los datos que pidió (nota como en la imagen las flechas van en ambos sentidos, porque el controlador pide datos, y el modelo responde con los datos solicitados).

Una vez el controlador tiene los datos del curso de AngularJS, se los manda a la vista, la vista aplica los estilos, organiza la información y construye la página que vez en el navegador.

Resumamos entonces los conceptos.

### MODELO

Se encarga de los datos, generalmente (pero no obligatoriamente) consultando la base de datos. Actualizaciones, consultas, búsquedas, etc. todo eso va aquí, en el modelo.

### CONTROLADOR

Se encarga de... controlar, recibe las órdenes del usuario y se encarga de solicitar los datos al modelo y de comunicárselos a la vista.

**VISTAS**

Son la representación visual de los datos, todo lo que tenga que ver con la interfaz gráfica va aquí. Ni el modelo ni el controlador se preocupan de cómo se verán los datos, esa responsabilidad es únicamente de la vista.

**Framework**

Un **framework** es un conjunto de componentes, fácilmente **reutilizables**, **escalables** y de **fácil mantenimiento**, de allí el anglicismo, **framework**(mesa de trabajo). Verás cuando estamos construyendo un sitio web existen **ciertos conjuntos de componentes** que necesitarás en casi cualquier implementación web:

- Autenticación de usuarios(registrarse, iniciar sesión, cerrar sesión).
- Panel de administración para tu sitio web(en esto Django se luce).
- Formularios.
- Una forma de subir archivos.

**”Entre otros muchos componentes, que si alguna vez has programado en PHP orientado a objetos, sabrás que se vuelven repetitivos y monótonos, de aplicación en aplicación. Ya sabes lo que vendría a ser el CRUD(Create, Read, Update, Delete o Crear, Leer, Actualizar, Eliminar).”**

Los **frameworks** existen para ahorrarte tener que **reinventar la rueda** y ayudarte a **aliviar la carga de trabajo** cuando construyes un sitio.

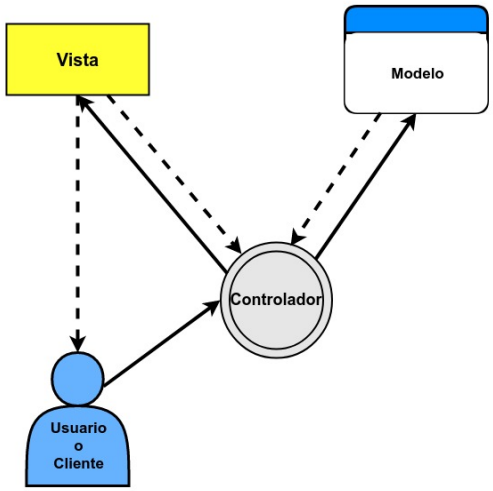
**¿Por qué usar un framework?**

Los **frameworks como Django** dejan a tu disposición una serie de herramientas y componentes, que reducen y mejoran de manera significativa el ritmo de trabajo de tu aplicación, **permitiendo de esta manera, concentrarse en áreas** que por su complejidad o naturaleza requieran un mayor carga de trabajo, delegando de esta manera el trabajo de desarrollo en áreas más específicas y dejando que el **framework** se encargue de las tareas más repetitivas o que por su simplicidad representen una carga innecesaria en el ritmo de trabajo.

**¿Y de qué va MTV, en todo esto?**

En la construcción de un sitio web, aplicación o programa, existen algo llamado patrones de diseño, que no son más que patrones para resolución de problemas en el desarrollo de tu aplicación.

Uno de los patrones de diseño más utilizados y más conocidos es el **MVC**(Modelo, Vista, Controlador), que propone lo siguiente:



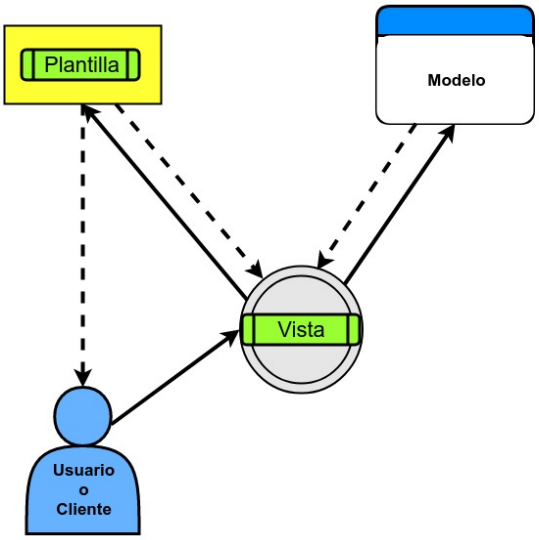

**M significa "Model" (Modelo),** la cual es una **capa de abstracción en código de la base de datos** manejada como un objeto a través del **framework**, el cual se encarga de la **comunicación e interacción con la base de datos.**

**V significa "View" (Vista),** a la que le designa nuestra **lógica de presentación(Frontend)**, la manera en la que se presentan y muestran, los datos extraídos de la base de datos a través del modelo, que hayan sido **requeridos por el controlador**.

**C significa "Controller"(Controlador),** la sección más reactiva de nuestra aplicación y sobre la cual se suele realizar la mayor parte de nuestro trabajo de **Backend**(nuestra **lógica de negocio**)**,** es el intermediario(**Middleware**) entre las solicitudes de usuario(cliente) a través de la vista y nuestros modelos. Se encarga de designar qué datos deben ser **pedidos al modelo y mostrados en la vista.**

**MVT**

El **MVC** es uno de los patrones de diseño más utilizados, por su versatilidad y simpleza, pero, rompiendo dicha convivencia, debido a que la "C" en **Django** es manejada por el mismo framework y la parte más importante se produce en los modelos, las plantillas y las vistas, Django es conocido como un Framework **MVT** donde:




**M significa “Model” (Modelo),** donde sigue siendo, la capa designada a la **interaccion y comunicacion,** con la información alojados en la base de datos.

**T significa “Template” (Plantilla),** en este caso la plantilla cumple la función de nuestra vista, la sección designada a la **lógica de negocios,** la representación de nuestros datos en código html.

**V significa “View” (Vista),** la **“C”** en el patrón de diseño **MTV,** es la capa designada a la **lógica de negocios**, a través de la que pasarán los datos del **modelo** a la **plantilla**.

### Microservicios

Los microservicios, o arquitectura de microservicios, es un enfoque para el desarrollo de software en el que una aplicación grande se construye como un conjunto de componentes o servicios modulares.

Cada módulo admite una tarea específica o un objetivo de negocio y utiliza una interfaz simple y bien definida, como una **interfaz de programación de aplicaciones (API)**, para comunicarse con otros conjuntos de servicios.

### Cómo funcionan los microservicios

En la arquitectura de microservicios, una aplicación se divide en servicios. Cada uno ejecuta un proceso único y generalmente administra su propia base de datos. Un servicio puede generar alertas, registrar datos, admitir **interfaces de usuario** (UI), manejar la identificación o autenticación del usuario y realizar otras tareas.

El paradigma de microservicio proporciona a los equipos de desarrollo un enfoque más descentralizado para construir software. Los microservicios permiten que cada servicio sea aislado, reconstruido, re desplegado y administrado de forma independiente. Por ejemplo, si un programa no genera informes correctamente, puede ser más fácil rastrear el problema a ese servicio específico. Ese servicio específico podría luego probarse, reiniciarse, parchearse y volver a implementarse según sea necesario, independientemente de otros servicios.

**Microservicios vs. arquitectura monolítica**

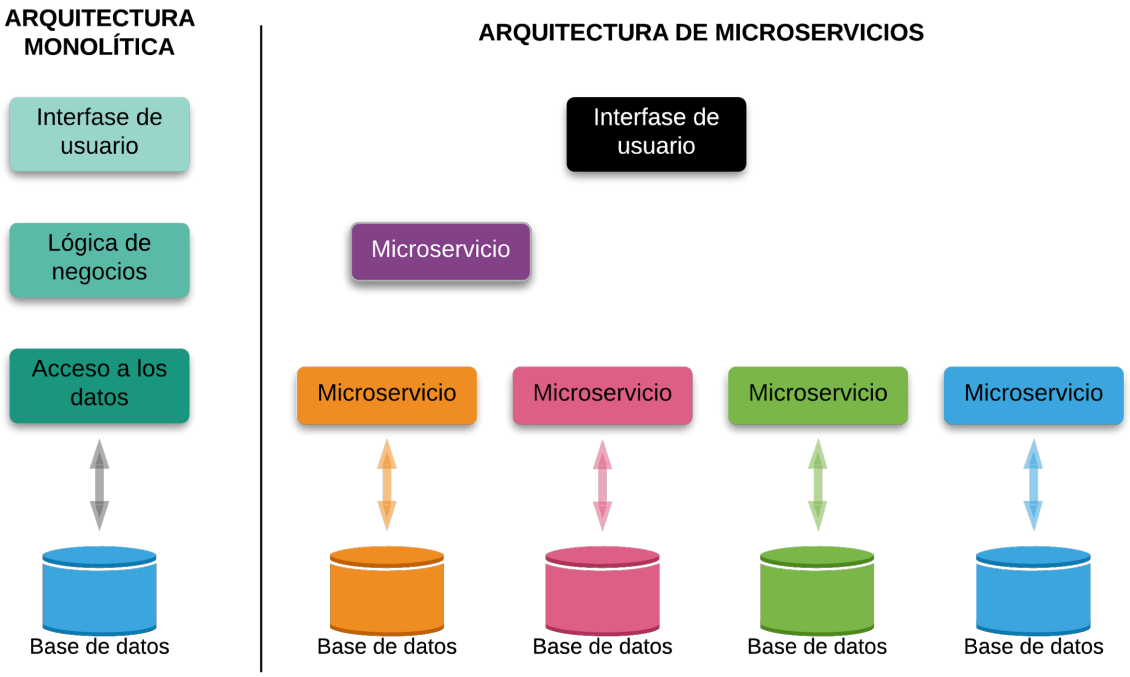
En una arquitectura monolítica, todo el código está en un archivo ejecutable principal, que puede ser más difícil de solucionar, probar y actualizar. Si hay un problema en una base de código, ese problema podría ubicarse en cualquier lugar dentro del software. Habría más pruebas, y éstas tardarían más debido a la cantidad de código monolítico involucrado. En una aplicación monolítica, cualquier pequeño cambio o actualización requiere compilar e implementar una versión completamente nueva de la aplicación. Esto significa que cualquier desarrollo de aplicaciones monolíticas implica una planificación, preparación, tiempo y gasto significativos.

Además, las aplicaciones monolíticas son más difíciles de escalar. Cuando una aplicación monolítica alcanza una limitación de su capacidad, como el rendimiento de datos o algún otro cuello de botella, la única alternativa práctica es implementar otra iteración completa de toda la aplicación monolítica: administrar el tráfico entre las instancias utilizando balanceadores de carga.

En comparación, es posible escalar solo los servicios de una aplicación de microservicio agregando instancias de contenedor de solo esos servicios. Esto hace que el microservicio de escala sea mucho más eficiente en recursos que las aplicaciones de escala utilizando una arquitectura monolítica.

Los microservicios facilitan la prueba y el despliegue de cambios. Debido a que cada uno está separado de los otros, se mejora el aislamiento de fallas. Si hay un problema en el software, el servicio problemático puede ser aislado, remediado, probado y redistribuido sin la necesidad de realizar una prueba de regresión de toda la aplicación como ocurre con las arquitecturas de aplicaciones monolíticas tradicionales. La arquitectura de microservicios mejora la agilidad empresarial con un desarrollo e implementación de software más rápido en comparación con la arquitectura de software monolítica.

Sin embargo, los microservicios no son libres de gestión. Con la misma cantidad de servicios que un producto de software monolítico, la administración requerida en una arquitectura de microservicios podría ser más compleja ya que cada servicio está separado uno del otro. Esto puede llevar a dificultades para manejar todas las partes de un todo. Por ejemplo, se necesita una cuidadosa supervisión y administración para rastrear la disponibilidad y el rendimiento de todos los servicios de componentes que operan dentro de una aplicación de microservicio.




### Microservicio pros y contras

La arquitectura de microservicios plantea una serie de concesiones para los desarrolladores de software. En términos de ventajas, se observan:

- Se despliegan fácilmente.
- Requieren menos tiempo de desarrollo.
- Puede escalar rápidamente
- Se puede reutilizar en diferentes proyectos.
- Contienen mejor aislamiento de fallas.
- Puede ser desplegado en equipos relativamente pequeños.
- Trabaja bien con los contenedores.

Sin embargo, también hay inconvenientes con la arquitectura de microservicios, tales como:

- Potencialmente demasiada granularidad.
- Esfuerzo extra de diseño para la comunicación entre servicios.
- Latencia durante el uso pesado.
- Pruebas complejas.

### Microservicios y DevOps

**DevOps** combina tareas entre la aplicación y los equipos de operaciones del sistema. Con el aumento de las comunicaciones entre los desarrolladores y el personal de operaciones, un equipo de TI puede crear y administrar mejor la infraestructura. Las operaciones de TI aseguran el presupuesto para capacidades, tareas operativas, actualizaciones y más. Los

desarrolladores y los equipos de aplicaciones pueden administrar bases de datos, servidores, software y hardware utilizados en la producción.

El trabajo en equipo y la colaboración entre los equipos de desarrollo y operaciones son necesarios para soportar el ciclo de vida de los microservicios, prestándose a los equipos de DevOps. También es la razón por la que los equipos de DevOps experimentados están bien equipados para emplear arquitectura de microservicios en proyectos de desarrollo de software.

### Arquitectura de microservicios vs. SOA

SOA es una arquitectura de software donde cada uno de sus servicios utiliza protocolos. Esto permite a los usuarios combinar funcionalidades y dar vida a aplicaciones creadas a partir de servicios anteriores. SOA ha sido la práctica de desarrollo estándar durante casi dos décadas. Sin embargo, el ingenio de SOA se pone en tela de juicio cuando se trabaja con la computación en la nube. Con la nube, SOA carece de escalabilidad y se ralentiza con los cambios de solicitud de trabajo, lo que limita el desarrollo de la aplicación.

Muchos desarrolladores consideran que la arquitectura de microservicios es un enfoque más granular de SOA. Los defensores del modelo SOA creen que la arquitectura de microservicios es la evolución natural de SOA necesaria para adaptarse a la **computación en la nube** y satisfacer las crecientes demandas de ciclos de desarrollo de software más rápidos.

Otros creen que los microservicios son un enfoque más independiente de la plataforma para el desarrollo de aplicaciones y, por lo tanto, deben tener un nombre único. Este grupo podría argumentar que SOA vive en las capas de administración de microservicio.

### Microservicios y contenedores

Un contenedor es un paquete de software individual y ejecutable, que incluye todas las dependencias que se necesitan para funcionar de manera independiente. Los contenedores están separados del resto del software que los rodea, y muchos contenedores pueden emplearse en el mismo entorno. En una arquitectura de microservicios, cada servicio se crea en contenedores individualmente en el mismo entorno, como el mismo servidor o los servidores relacionados.

Una máquina virtual (VM) se puede utilizar como alternativa a los contenedores para crear microservicios. Una máquina virtual simula los sistemas informáticos para producir funcionalidades de una computadora física. Cada servicio podría potencialmente utilizar una máquina virtual para alojar una característica prevista. Sin embargo, las máquinas virtuales a menudo se evitan para los microservicios debido al sistema operativo individual (SO) y otros gastos generales necesarios para cada máquina virtual. Los contenedores son mucho más eficientes en el uso de recursos porque solo se requieren el código subyacente y las dependencias relacionadas para operar el servicio.

### Seguridad de la arquitectura de microservicios

La arquitectura de microservicios puede aliviar algunos problemas de seguridad que surgen con las aplicaciones monolíticas. Los microservicios simplifican el monitoreo de seguridad porque las diversas partes de una aplicación están aisladas. Una brecha de seguridad

podría ocurrir en una sección sin afectar otras áreas del proyecto. Los microservicios ofrecen resistencia contra los ataques distribuidos de denegación de servicio (DDoS) cuando se usan con contenedores al minimizar una toma de control de la infraestructura con demasiadas solicitudes de servidor.

Sin embargo, todavía existen desafíos al proteger aplicaciones de microservicios, que incluyen:

- Más áreas de red están abiertas a vulnerabilidades.
- Una menor coherencia general entre las actualizaciones de la aplicación permite más violaciones de seguridad.
- Hay una mayor área de ataque, a través de múltiples puertos y **APIs**.
- Hay una falta de control de software de terceros.
- La seguridad debe mantenerse para cada servicio.

Los desarrolladores de microservicios han ideado estrategias para aliviar los problemas de seguridad. Para ser proactivo, use un escáner de seguridad, utilice limitaciones de control de acceso, redes internas seguras, incluidos los entornos Docker, y opere fuera de los silos, comunicándose con todas las partes de la operación.

**Despliegue de aplicaciones de microservicio**

Se han producido tres avances principales para hacer viable la arquitectura de microservicios.

El primer avance, los contenedores, permite un medio consistente y eficiente de recursos para empaquetar servicios individuales. Docker es una herramienta popular para desarrolladores, que permite el uso de contenedores en las instalaciones o en la nube pública o privada. Esto ofrece una amplia variedad de alternativas de implementación para desarrolladores de microservicios y empresas.

El segundo desarrollo importante es la aparición de herramientas de orquestación, como Kubernetes. Estas herramientas ayudan a automatizar el escalado, la implementación y la administración de contenedores.

Un tercer avance importante para los microservicios es la evolución de la malla. Una malla de servicios es una capa de infraestructura dedicada a la comunicación entre servicios individuales. Las mallas de servicio hacen que estas comunicaciones sean más rápidas, más seguras, visibles y confiables.

Cuando cientos de servicios se comunican entre sí, se vuelve complicado saber cuáles interactúan entre sí. Linkerd es una herramienta de orquestación que logra comunicaciones seguras, rápidas y visibles entre los servicios mediante la captura de comportamientos, como el equilibrio de carga compatible con la latencia o el descubrimiento del servicio.

Los niveles de complejidad en los microservicios están disminuyendo constantemente debido a estos avances, como las complejidades de la supervisión y el registro, a medida que más organizaciones y equipos de desarrollo adoptan los microservicios.