Introducción a arquitecturas Web API y MVC

Cualquier aplicación funcional es lo suficientemente grande como para requerir una estructura que la haga fácil de comprender para poder ser desarrollada y mantenida. Las estructuras no solo son recomendadas sino que son necesarias.

Existen una amplia variedad de estructuras para aplicaciones ya sea para las más pequeñas o para estructuras monstruosas. En esta unidad nos vamos a detener a analizar las estructuras más frecuentes que se pueden hallar en el mercado.

Es importante destacar que no en todas las ocasiones estas estructuras se hallan de forma pura, es decir por lo general las aplicaciones combinan estructuras para lograr obtener ventajas de las que la componen

Arquitecturas por capas

Una capa es un conjunto de “cosas” que tienen cierta responsabilidad. Por ejemplo, una capa puede ser un conjunto de clases, agrupadas en un paquete, dentro de nuestro programa que representan cierta responsabilidad o también puede ser un ejecutable que se comunica con otros y cada ejecutable representa una capa o pueden ser un sistema y cada sistema es en sí una capa. Cuando nos referimos a capas es una abstracción de responsabilidades.

Además, la arquitectura establece reglas de cómo se deben comunicar las capas.

Primera regla

Cada capa debe tener una responsabilidad única. Es decir que las capas deben estar perfectamente delimitadas de que se ocupa cada una de ellas, por ejemplo, podemos tener una capa de “presentación” que será la encargada de atender los eventos del cliente y encargada de representar la información para el mismo. Por otro lado, podemos tener la capa de “acceso a datos” que será la encargada de guardar y acceder a los datos.

Segunda regla

Las capas deben respetar una estructura jerárquica estricta. Quiere decir que cada capa puede comunicarse sólo con la que está debajo suyo, pero NO al revés. Por ejemplo, una clase ubicada en la capa de presentación puede llamar a un método ubicado en la capa de acceso a datos, pero nunca la capa de acceso a datos puede llamar a un método de la capa de presentación. Y cuando nos referimos a la próxima más baja significa que no se puede saltar capas. Veamos un ejemplo gráfico.

Para este ejemplo utilizaremos la separación de tres capas “presentación”, “lógica de negocio” y “acceso a datos”. Esto es lo que se debería hacer y lo que no se debería hacer.

Ventajas

Entre sus ventajas se encuentran las siguientes:

- Es fácil testear cada capa por separado debido a la separación clara de responsabilidades que existe entre ellas.
- Al momento de hacer un cambio, si se implementó bien la separación de responsabilidad, este cambio solo debe impactar a la capa responsable y no a todas. Esto se conoce como desacople.

Desventajas

- Si se implementaron demasiadas capas el rendimiento de la aplicación puede verse afectado
- Ciertas operaciones al ser modificadas pueden afectar a todas las capas, haciendo visible que no existe un 100% de desacople entre estas.

Arquitectura de tres capas

Esta arquitectura permite desarrollar n número de capas siempre y cuando se respeten las dos reglas expuestas. Sin embargo la arquitectura de 3 capas es bastante utilizada y es muy útil, pedagógicamente, para entender la separación de responsabilidades de las capas.

La arquitectura de tres capas en particular es muy utilizada ya que propone dividir las capas en 3 específicas que son frecuentes en aplicaciones empresariales web. Como veníamos mencionando en los ejemplos las capas que propone son las siguientes.

Presentación

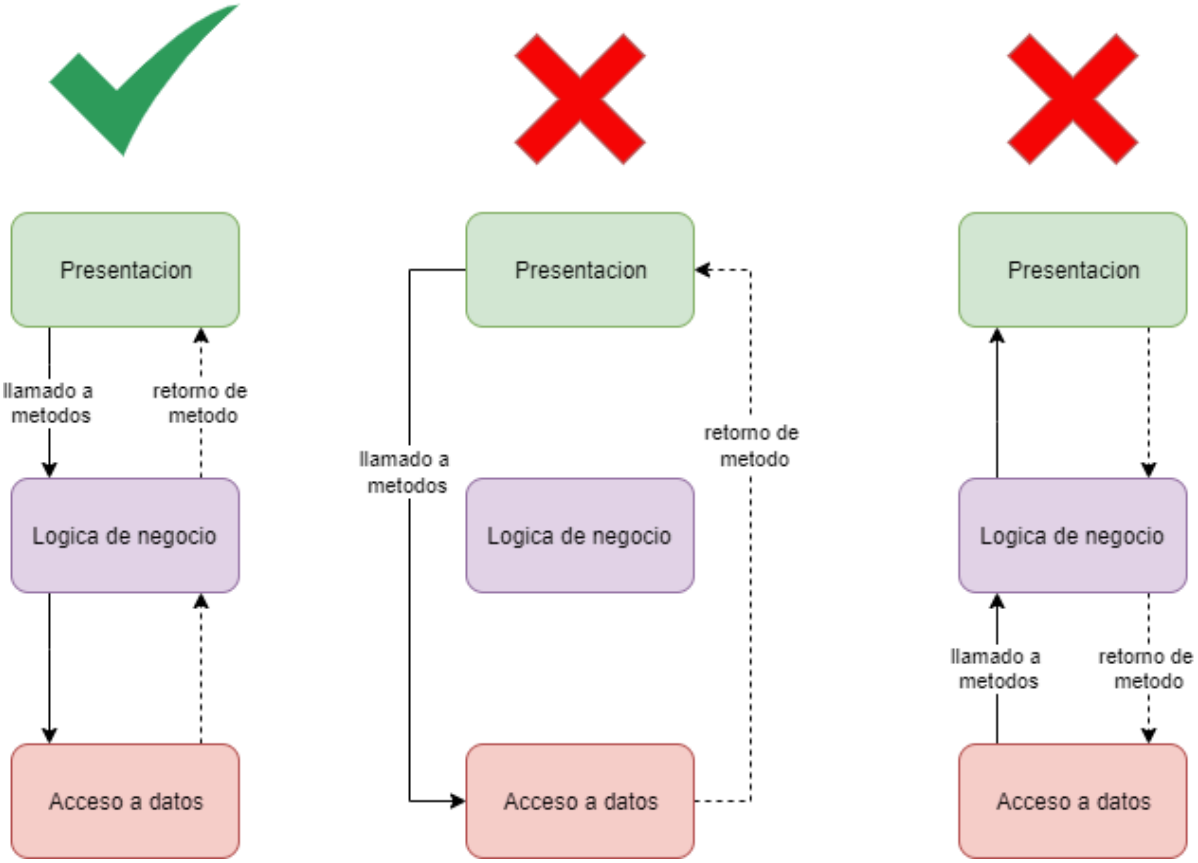
Atiende los eventos del cliente y representa los datos para el mismo. Teniendo en cuenta que el cliente puede ser un humano u otro sistema, esta capa será encargada en caso del humano de atender los clics (u otros eventos) en un HTML y de renderizar la información de manera visual. En caso de que sea otro sistema puede atender peticiones rest(daremos esto en los siguientes capítulos) y devolver información en un formato estructurado (json, xml, etc) que es más fácil de interpretar por un sistema.

Lógica de negocio

En esta capa se encuentra todo lo que refiere a las reglas que se encuentran en el negocio, o sea los requerimientos funcionales de nuestro sistema. Por ejemplo, si se tiene un alta de usuario esta capa debe proveer el medio para altaDeUsuario y dentro del método se debe realizar todos los pasos para dar de alta un usuario (enviar mail, validar nombre, etc).

Acceso a datos

Mediante esta capa podremos obtener o guardar los datos que utilizará nuestra aplicación. Observar que no habla de cómo se realiza la persistencia, si es en base de datos o en archivo o etc, solo habla de acceso a datos. Por ejemplo, esta capa debería proveer un medio para poder guardar el usuario y otro para obtenerlo.



Beneficio

Si logramos realizar esta separación de responsabilidades podemos notar como a una capa no le interesa cómo está implementada la otra. Así yo puedo reemplazar las implementaciones de las capas, pero esto no afectaría a las demás. Ejemplo: mi capa de acceso a datos ofrece un método guardarUsuario y por dentro este es guardado en un txt. En el futuro deseo cambiar esta implementación entonces seguiría existiendo el método guardarUsuario pero por dentro este es persistido en una base de datos. Para la capa de

lógica de negocio esto no debería importar ya que lo único que desea es que el usuario sea guardado sin importar cómo.

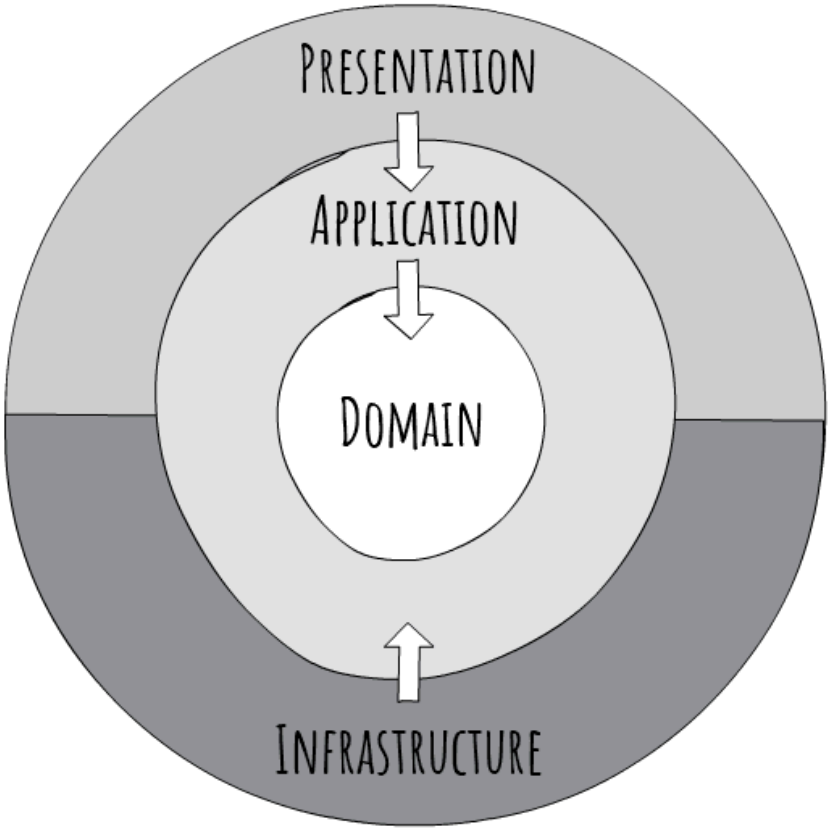
Esto fue todo sobre arquitectura de capas y en especial de tres capas. Próximamente veremos otras arquitecturas para comparar qué beneficios posee cada una.

Clean architecture

Descripción del patrón

En Clean architecture, la capa de dominio y aplicación son el centro del diseño y se conocen como el core del sistema.
La capa de dominio contiene lógica empresarial y la capa de aplicación contiene lógica de negocio. La diferencia está en que la lógica empresarial se puede compartir entre múltiples sistemas y la lógica de negocio corresponde a un sistema en particular.
El core no debe depender de el acceso a la base de datos ni a otras cuestiones de infraestructura, por lo tanto las dependencias son invertidas.
Esto se obtiene agregando interfaces o abstracciones en el core, que son implementadas en las capas exteriores. Por ejemplo, si se quiere implementar el patrón Repositorio, se debe definir la interfaz dentro del core y la implementación se hace en la capa de infraestructura.
Todas las dependencias apuntan hacia el core, y el core no tiene dependencia de ninguna otra capa. Las capas de infraestructura y presentación dependen del core, pero no la una de la otra.

Representación gráfica



Estructura básica de carpetas en cada proyecto

Esta es una estructura sugerida para ordenar el código. La misma puede variar según criterio del arquitecto de software y de los patrones que se deseen implementar en la solución.

- Domain
    - Entities: Entidades del modelo de dominio.
    - Enums: Enumeraciones.
    - Exceptions: Custom Exceptions.
    - Interfaces (Según criterio del arquitecto): Interfaces de repositorios u otras clases que se podrían compartir con otros sistemas de la organización.

- Application
    - Services: Servicios que orquestan las clases necesarias para cumplir con las request (casos de uso). Dependen de abstracciones y nunca de clases de librerías externas.

- Models: Dtos tanto para requests como para responses.

- Interfaces: Interfaces de los servicios de la capa de Application e Infrastructure. También se puede incluir las interfaces de los repositorios.

- Infrastructure
  - Data: Clases para implementar el patrón repositorio. Clases de acceso a datos, como el Context.
    - Migrations: Clases correspondientes a las migraciones y snapshot de la base de datos. (Para entity framework)

  - Services: Clases correspondiente a servicios que cumplen una función específica y generalmente utilizan librerías externas.

- Presentation (Web o API)
  - Controllers: Objetos controllers, que se encargan de recibir la request, hacer el data binding, data validation y mapear el action method correspondiente, para finalmente generar una response.

Código de ejemplo

https://github.com/jasontaylordev/CleanArchitecture
https://github.com/ardalis/CleanArchitecture

Recursos

https://jasontaylor.dev/clean-architecture-getting-started/

Web API

El término API es una abreviatura de Application Programming Interfaces, que en español significa interfaz de programación de aplicaciones. Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Así pues, podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores de terceros.

Una de las principales funciones de las API es poder facilitar el trabajo a los desarrolladores y ahorrarles tiempo y dinero. Por ejemplo, si estás creando una aplicación que es una tienda online, no necesitarás crear desde cero un sistema de pagos u otro para verificar si hay stock disponible de un producto. Podrás utilizar la API de un servicio de pago ya existente,

por ejemplo PayPal, y pedirle a tu distribuidor una API que te permita saber el stock que ellos tienen.
Con ello, no será necesario tener que reinventar la rueda con cada servicio que se crea, ya que podrás utilizar piezas o funciones que otros ya han creado. Imagínate que cada tienda online tuviera que tener su propio sistema de pago, para los usuarios normales es mucho más cómodo poder hacerlo con los principales servicios que casi todos utilizan.
También son útiles para cuando lo único que se quiere es utilizar deliberadamente las funciones de determinado servicio para ofrecer ventajas a sus usuarios o atraer a los usuarios de ese servicio a que utilicen tu aplicación.
Por ejemplo, piensa que quieres crear una aplicación que se conecte a las publicaciones que hay en Twitter, pues para ello tendrás que conectar tu aplicación al servicio mediante la API que Twitter tiene disponible para los desarrolladores.

Request y Response

HTTP REQUEST y RESPONSE son dos conceptos básicos en el desarrollo web, pero no siempre le quedan claros a los programadores.
HTTP significa HyperText Transfer Protocol. Esta es la forma de comunicación de datos básica en Internet. La comunicación de datos empieza con un request enviado del cliente, y termina con la respuesta del servidor web.
Por ejemplo, si fuera un ejemplo clásico con un ser humano visitando una página Web:

1. Un sitio web que empieza con la URL http:// es entrado en un navegador web de la computadora del cliente. El navegador puede ser Chrome, Firefox, o Internet explorer, no importa.
2. El navegador envía un request al servidor web que está hospedado en el website.

3. El servidor web regresa una respuesta como un página de HTML, o algún otro formato de documento al navegador (puede ser un mp4, mp3, pdf, doc, entre otros soportados por el navegador)

4. El navegador despliega el response del servidor al usuario. Por supuesto esto dependerá de los formatos que soporte el navegador.

No todas las requests se hacen desde los navegadores de usuarios finales, de hecho muchas peticiones se hacen desde aplicaciones(Discord, Steam, Whatsapp, Instagram) o incluso de dispositivos IoT(Google Home, Alarmas).

De esta forma mucha de la comunicación vía web, ya no se hace directamente entre humano-máquina, cada vez más son aplicaciones automatizadas de ambos lados, las que envían datos. Es decir por ejemplo yo tengo un programa en PHP, APEX, o algún otro lenguaje que utilizo para enviar request y recibir response, de manera que la comunicación es de máquina a máquina

Las peticiones o requests se hacen siempre a un endpoint. Es decir un punto de acceso de una API que procesa dicha solicitud y responde de acuerdo a su implementación interna. Aquí se puede ver que las APIs toman claramente la encapsulación de la POO

Ahora bien, ¿Qué contienen el request y el response?

HTTP Request Structure from Client

Un HTTP request se compone de:

- Método: GET, POST, PUT, etc. Indica que tipo de request es.
- Path: la URL que se solicita, donde se encuentra el resource.
- Protocolo: contiene HTTP y su versión, actualmente 1.1.
- Headers. Son esquemas de key: value que contienen información sobre el HTTP request y el navegador. Aquí también se encuentran los datos de las cookies. La mayoría de los headers son opcionales.
- Body. Si se envía información al servidor a través de POST o PUT, ésta va en el body.

HTTP Response Structure from Web Server
Una vez que el navegador envía el HTTP request, el servidor responde con un HTTP response, compuesto por:

- Protocolo. Contiene HTTP y su versión, actualmente 1.1.
- Status code. El código de respuesta, por ejemplo: 200 OK, que significa que el GET request ha sido satisfactorio y el servidor devolverá los contenidos del documento solicitado. Otro ejemplo es 404 Not Found, el servidor no ha encontrado el resource solicitado.
- Headers. Contienen información sobre el software del servidor, cuando se modificó por última vez el resource solicitado, el mime type, etc. De nuevo la mayoría son opcionales.
- Body. Si el servidor devuelve información que no sean headers ésta va en el body.

Verbos HTTP

También conocidos como métodos HTTP. Vimos como una request contiene un Request Method, pues bien eso es un verbo HTTP. Indican que tipo de petición queremos hacer al servidor.

Hay distintos tipos de peticiones que podemos hacer hacia el servidor mismo.

Estos verbos indican qué acción queremos realizar sobre el servidor y son GET, POST, PUT, PATCH, DELETE, HEAD, CONNECT, OPTIONS y TRACE. Cada uno indica una acción diferente a la que el servidor debe responder.

A continuación haremos una pequeña descripción de los más importantes.

GET
El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.
POST
El método POST se utiliza para enviar una entidad a un recurso en específico. Se uso más frecuente es para crear un nuevo recurso.

PUT
El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición. Es para hacer una modificación total de un recurso.

DELETE: El método DELETE borra un recurso en específico.

PATCH: El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

Model View Controller (MVC)

En líneas generales, MVC es una propuesta de arquitectura del software utilizada para separar el código por sus distintas responsabilidades, manteniendo distintas capas que se encargan de hacer una tarea muy concreta, lo que ofrece beneficios diversos.

MVC se usa inicialmente en sistemas donde se requiere el uso de interfaces de usuario, aunque en la práctica el mismo patrón de arquitectura se puede utilizar para distintos tipos de aplicaciones. Surge de la necesidad de crear software más robusto con un ciclo de vida más adecuado, donde se potencie la facilidad de mantenimiento, reutilización del código y la separación de conceptos.

Su fundamento es la separación del código en tres capas diferentes, acotadas por su responsabilidad, en lo que se llaman Modelos, Vistas y Controladores, o lo que es lo mismo, Model, Views & Controllers, si lo prefieres en inglés. En este artículo estudiaremos con detalle estos conceptos, así como las ventajas de ponerlos en marcha cuando desarrollamos.

MVC es un "invento" que ya tiene varias décadas y fue presentado incluso antes de la aparición de la Web. No obstante, en los últimos años ha ganado mucha fuerza y seguidores gracias a la aparición de numerosos frameworks de desarrollo web que utilizan el patrón MVC como modelo para la arquitectura de las aplicaciones web.

Nota: Como ya hemos mencionado, MVC es útil para cualquier desarrollo en el que intervengan interfaces de usuario. Sin embargo, a lo largo de este artículo explicaremos el paradigma bajo el prisma del desarrollo web.

Por qué MVC

La rama de la ingeniería del software se preocupa por crear procesos que aseguren calidad en los programas que se realizan y esa calidad atiende a diversos parámetros que son deseables para todo desarrollo, como la estructuración de los programas o reutilización del código, lo que debe influir positivamente en la facilidad de desarrollo y el mantenimiento. Los ingenieros del software se dedican a estudiar de qué manera se pueden mejorar los procesos de creación de software y una de las soluciones a las que han llegado es la arquitectura basada en capas que separan el código en función de sus responsabilidades o conceptos. Por tanto, cuando estudiamos MVC lo primero que tenemos que saber es que está ahí para ayudarnos a crear aplicaciones con mayor calidad.

Quizás, para que a todos nos queden claras las ventajas del MVC podamos echar mano de unos cuantos ejemplos:

- Si queremos que en un equipo intervengan perfiles distintos de profesionales y trabajen de manera autónoma, como diseñadores o programadores, ambos tienen que tocar los mismos archivos y el diseñador se tiene necesariamente que relacionar con mucho código en un lenguaje de programación que puede no serle familiar, siendo que a éste quizás solo le interesan los bloques donde hay HTML. De nuevo, sería mucho más fácil la separación del código.
- Durante la manipulación de datos en una aplicación es posible que estemos accediendo a los mismos datos en lugares distintos. Por ejemplo, podemos acceder a los datos de un artículo desde la página donde se muestra éste, la página donde se listan los artículos de un manual o la página de backend donde se administran los artículos de un sitio web. Si un día cambiamos los datos de los artículos (alteramos la tabla para añadir nuevos campos o cambiar los existentes porque las necesidades de nuestros artículos varían), estamos obligados a cambiar, página a página, todos los lugares donde se consumían datos de los artículos. Además, si tenemos el código de acceso a datos disperso por decenas de lugares, es posible que estemos repitiendo las mismas sentencias de acceso a esos datos y por tanto no estamos reutilizando código.

Capas

Modelos

Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.

No obstante, cabe mencionar que cuando se trabaja con MVC lo habitual también es utilizar otras librerías o algún ORM como Entity Framework, que nos permiten trabajar con abstracción de bases de datos y persistencia en objetos. Por ello, en vez de usar directamente sentencias SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.

Vistas

Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra aplicación en HTML. En las vistas nada más tenemos los códigos HTML y PHP que nos permite mostrar la salida.

En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.

Controladores

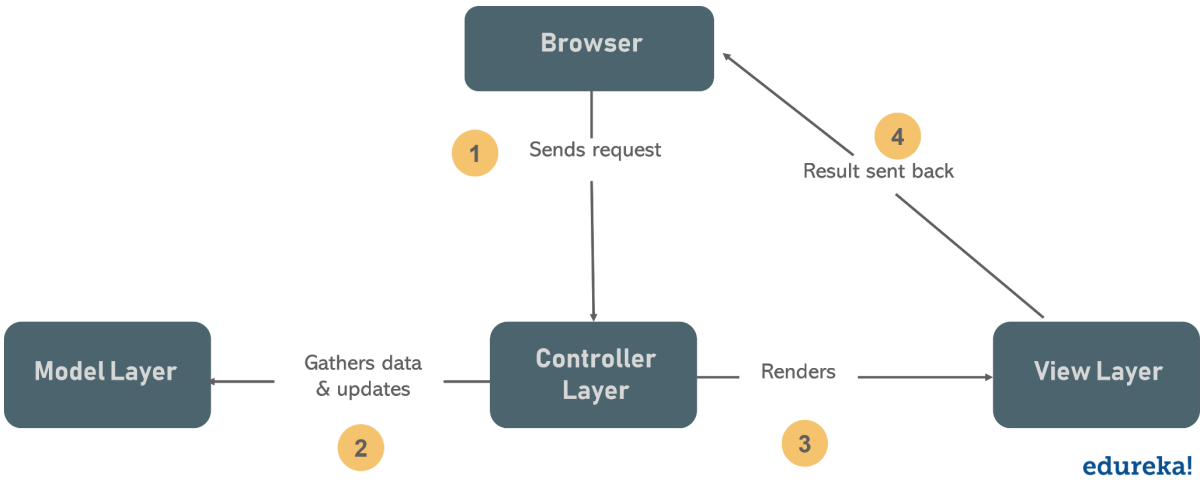
Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc.

En realidad es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

Arquitectura de aplicaciones MVC

A continuación encontrarás un diagrama que te servirá para entender un poco mejor cómo colaboran las distintas capas que componen la arquitectura de desarrollo de software en el patrón MVC.

En esta imagen hemos representado con flechas los modos de colaboración entre los distintos elementos que formarían una aplicación MVC, junto con el usuario. Como se puede ver, los controladores, con su lógica de negocio, hacen de puente entre los modelos y las vistas. Pero además en algunos casos los modelos pueden enviar datos a las vistas. Veamos paso a paso cómo sería el flujo de trabajo característico en un esquema MVC.



El usuario(a través del browser) realiza una solicitud a nuestro sitio web. Generalmente estará desencadenada por acceder a una página de nuestro sitio. Esa solicitud le llega al controlador.
El controlador se comunica tanto con modelos como con vistas. A los modelos les solicita datos o les manda realizar actualizaciones de los datos. A las vistas les solicita la salida correspondiente, una vez se hayan realizado las operaciones pertinentes según la lógica del negocio.
Para producir la salida, en ocasiones las vistas pueden solicitar más información a los modelos. En ocasiones, el controlador será el responsable de solicitar todos los datos a los modelos y de enviarlos a las vistas, haciendo de puente entre unos y otros. Sería corriente tanto una cosa como la otra, todo depende de nuestra implementación; por eso esa flecha la hemos coloreado de otro color.

Lógica de negocio / Lógica de la aplicación

Hay un concepto que se usa mucho cuando se explica el MVC que es la **"lógica de negocio"**. Es un conjunto de reglas que se siguen en el software para reaccionar ante distintas situaciones. En una aplicación el usuario se comunica con el sistema por medio de una interfaz, pero cuando acciona esa interfaz para realizar acciones con el programa, se ejecutan una serie de procesos que se conocen como la lógica del negocio. Este es un concepto de desarrollo de software en general.

La lógica del negocio, aparte de marcar un comportamiento cuando ocurren cosas dentro de un software, también tiene normas sobre lo que se puede hacer y lo que no se puede hacer. Eso también se conoce como reglas del negocio. Bien, pues en el MVC la lógica del negocio queda del lado de los modelos. Ellos son los que deben saber cómo operar en diversas situaciones y las cosas que pueden permitir que ocurran en el proceso de ejecución de una aplicación. Es decir que en MVC gran parte de la lógica se maneja fuera del controlador, esto puede ser en la capa de modelos o bien creando una nueva capa con la lógica de negocio(Esto la diferencia de la capa de datos en la arquitectura de 3 capas)

Por ejemplo, pensemos en un sistema que implementa usuarios. Los usuarios pueden realizar comentarios. Pues si en un modelo nos piden eliminar un usuario nosotros debemos

borrar todos los comentarios que ha realizado ese usuario también. Esto no es una responsabilidad del controlador y forma parte de lo que se llama la lógica del negocio y se ejecutará en la capa que hayamos definido.

Sin embargo existe otro concepto que se usa en la terminología del MVC que es la **"lógica de aplicación"**, que es algo que pertenece a los controladores. Por ejemplo, cuando me piden ver el resumen de datos de un usuario. Esa acción le llega al controlador, que tendrá que acceder al modelo del usuario para pedir sus datos. Luego llamará a la vista apropiada para poder mostrar esos datos del usuario. Si en el resumen del usuario queremos mostrar los artículos que ha publicado dentro de la aplicación, quizás el controlador tendrá que llamar al modelo de artículos, pedirle todos los publicados por ese usuario y con ese listado de artículos invocar a la vista correspondiente para mostrarlos. Todo ese conjunto de acciones que se realizan invocando métodos de los modelos y mandando datos a las vistas forman parte de la lógica de la aplicación.

En resumen, en MVC la lógica se divide en dos capas distintas, cierta lógica (lógica de negocios) está presente en la capa de Modelos o en la capa que hayamos definido para ella y cierta lógica en la capa de controladores(lógica de aplicación). Es importante destacar que en la capa de vistas nunca hay ningún tipo de lógica.