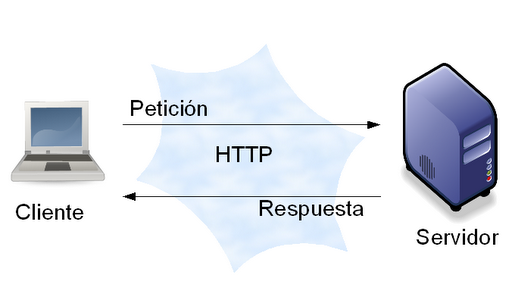
**Arquitectura**

Basados en la Arquitectura Cliente/Servidor, del libro de Maximiliano Cristiá (Estilos Arquitectónicos), donde nos dice que usualmente la comunicación entre componentes es de a pares y la inicia un cliente; al pedido de un servicio de un cliente le corresponde la respuesta del servidor respectivo. En otras palabras la comunicación entre cliente y servidor es, por lo general, asimétrica, por tal motivo definimos a la misma como:

**Descripción**

El estilo cliente/servidor define una relación entre dos aplicaciones en las cuales una de ellas (cliente) envía peticiones a la otra (servidor fuente de datos).

**Características**

* Es un estilo para sistemas distribuidos.
* Divide el sistema en una aplicación cliente, una aplicación servidora y una red que las conecta.
* Describe una relación entre el cliente y el servidor en la cual el primero realiza peticiones y el segundo envía respuestas.
* Puede usar un amplio rango de protocolos y formatos de datos para comunicar la información.

**Principios Clave**

* El cliente realiza una o más peticiones, espera por las respuestas y las procesa a su llegada.
* El cliente normalmente se conecta solo a uno o a un número reducido de servidores al mismo tiempo.
* El cliente interactúa directamente con el usuario, por ejemplo a través de una interfaz gráfica.
* El servidor no realiza ninguna petición al cliente.
* El servidor envía los datos en respuesta a las peticiones realizadas por los clientes conectados.
* El servidor normalmente autentifica y verifica primero al usuario y después procesa la petición y envía los resultados.

**Beneficios**

* Más seguridad ya que los datos se almacenan en el servidor que generalmente ofrece más control sobre la seguridad.
* Acceso centralizado a los datos que están almacenados en el servidor lo que facilita su acceso y actualización.
* Facilidad de mantenimiento ya que los roles y las responsabilidades se distribuyen entre los distintos servidores a través de la red lo que permite que un cliente no se vea afectado por un error en un servidor particular.

**Beneficios de uso de Capas**

* El mantenimiento de mejoras en una solución será mucho más fácil porque las funciones están localizadas y además las capas deben estar débilmente acopladas entre ellas y con alta cohesión internamente, lo cual posibilita variar de una forma sencilla diferentes implementaciones/combinaciones de capas.
* Otras soluciones deberían poder reutilizar funcionalidad expuesta por las diferentes capas, especialmente si se han diseñado para ello.
* Los desarrollos distribuidos son mucho más sencillos de implementar si el trabajo se ha distribuido previamente en diferentes capas lógicas.
* La distribución de capas (layers) en diferentes niveles físicos (tiers) pueden, en algunos casos, mejorar la escalabilidad. Aunque este punto hay que evaluarlo con cuidado, pues puede impactar negativamente en el rendimiento.

Actualmente, ASP.NET soporta tres modelos de programación: ASP.NET Web Forms, ASP.NET MVC y ASP.NET Web Pages. Aunque los tres modelos de programación se ejecutan sobre la misma base de ASP.NET, cada uno de ellos estructura la aplicación de maneras completamente distintas, promueve metodologías de desarrollo diferentes y se adapta a perfiles de desarrolladores distintos. Algunas características que son virtudes en unos modelos de programación, pueden ser consideradas debilidades en el otro.

Consultorio Virtual se basa en uno de estos: ASP.NET MVC ya que proporciona un modelo de programación basado en el popular patrón de arquitectura MVC. Entre sus principales características destacan su completa integración con pruebas unitarias y su separación más clara entre la lógica de presentación, la lógica de negocio y la lógica de acceso a datos.

**Capas y Conectores**

**- Capa de Presentación**

Esta capa es responsable de mostrar información al usuario e interpretar sus acciones.

Los componentes de las capas de presentación implementan por lo tanto la funcionalidad requerida para que los usuarios interactúen con la aplicación.

Normalmente es recomendable subdividir dichos componentes en varias sub-capas y aplicando patrones de tipo MVC ó MVP ó M-V-VM:

**Subcapa de Componentes Visuales (Vistas):** Estos componentes proporcionan el mecanismo base para que el usuario utilice la aplicación. Por lo tanto, son componentes que formatean datos en cuanto a tipos de letras y controles visuales, y también reciben datos proporcionados por el usuario.

**Subcapa de Controladores:** Para ayudar a sincronizar y orquestar las interacciones del usuario, puede ser útil conducir el proceso utilizando componentes separados de los componentes propiamente gráficos. Esto impide que el flujo de proceso y lógica de gestión de estados esté programada dentro de los propios controles y formularios visuales y permite reutilizar dicha lógica y patrones desde otros interfaces o „vistas‟. También es muy útil para poder realizar pruebas unitarias de la lógica de presentación. Estos “Controllers‟ son típicos de los patrones MVC y derivados.

**- Capa de Aplicación**

Esta capa forma parte de la propuesta de arquitecturas orientadas al Dominio.

Define los trabajos que la aplicación como tal debe de realizar y re-dirige a los objetos del dominio que son los que internamente deben de resolver los problemas.

Esta capa debe ser una capa „delgada‟, no debe contener realmente reglas del dominio o conocimiento de la lógica de negocio, simplemente debe coordinar tareas necesarias para la aplicación (software o aplicación como tal), aspectos necesarios para optimizaciones de la aplicación o tareas de la aplicación, no lógica de negocio/dominio y delegar posteriormente trabajo a los objetos del dominio (siguiente capa). Tampoco debe contener estados que reflejen la situación de la lógica de negocio interna pero sí puede tener estados que reflejen el progreso de una tarea de la aplicación a ser mostrada al usuario.

Es una capa en algunos sentidos parecida a las capas “Fachada de Negocio”, pues

en definitiva hará de fachada del modelo de Dominio, pero no solamente se encarga de simplificar el acceso al Dominio, hace algo más. Aspectos a incluir en esta capa serían:

* Agrupaciones/agregaciones de datos de diferentes entidades para ser enviadas de una forma más eficiente (minimizar las llamadas remotas), por la capa superior de servicios web. A esto se le llama DTO (Data Transfer Object).
* Acciones que consolidan o agrupan operaciones del Dominio dependiendo de las acciones mostradas en el interfaz mostrado al usuario.
* Mantenimiento de estados relativos a la aplicación (no estados internos del Dominio).
* Coordinación de acciones entre el Dominio y algunos aspectos de infraestructura, como por ejemplo, la acción de realizar una transferencia bancaria y que al mismo tiempo mande un e-mail a la partes interesada (siempre y cuando dicho envío de mail se considere una acción de la aplicación y no lógica intrínseca del Dominio). En cualquier caso, dicha agrupación de diferentes acciones es lo que haríamos en la capa de Aplicación (en el ejemplo, agruparíamos persistencia de datos pasando por el dominio y el envío de un e-mail realizado por la infraestructura). Pero solo se realizaría aquí la agrupación de dichas acciones. La realización de las acciones en si (persistencia y/o envío de e-mail del ejemplo) se realizarán en las capas inferiores.

**- Capa del Dominio**

Esta capa debe ser responsable de representar conceptos de negocio, información sobre la situación de los procesos de negocio e implementación de las reglas del dominio. También debe contener los estados que reflejan la situación de los procesos de negocio, aun cuando los detalles técnicos de persistencia se delegan a las capas de infraestructura (Repositorios, etc.).

Siguiendo los patrones de Arquitecturas N-Layer con Orientación al Dominio, esta capa tiene que ignorar completamente los detalles de persistencia de datos. Estas tareas de persistencia deben ser realizadas por las capas de infraestructura.

Normalmente podemos definir los siguientes elementos dentro de la capa de

Dominio:

**- “Entidades del Dominio‟:** Estos objetos son entidades de datos desconectados y se utilizan para obtener y transferir datos de entidades entre las diferentes capas. Adicionalmente es recomendable que contengan también en las mismas clases lógica del dominio relativo al contenido de entidad, por ejemplo, validaciones de datos, campos pre-calculados, relaciones con otras subentidades, etc. Estos datos representan al fin y al cabo entidades de negocio del mundo real, como productos o pedidos. Las entidades de datos que la aplicación utiliza internamente, son en cambio, objetos en memoria con datos y cierta lógica relacionada, como puedan ser clases propias, clases del framework o simplemente streams XML (aunque este último caso o clases con „solo datos‟ no dispondrían de lógica propia de la entidad y estaríamos cayendo en el llamado “Anemic Domain Model”). Siguiendo los patrones y principios recomendados, es bueno por ejemplo hacer uso de objetos POCO (Plain Old CLR Objects), es decir, de clases independientes con código completamente bajo nuestro control y situando estas entidades dentro del dominio, puesto que son entes del dominio e independientes de cualquier tecnología de infraestructura (persistencia de datos, ORMs, etc.). En cualquier caso, las entidades serán objetos flotantes a lo largo de toda o casi toda la arquitectura.

Relativo a DDD, y de acuerdo con la definición de Eric Evans, “Un objeto primariamente definido por su identidad se le denomina Entidad”. Las entidades son fundamentales en el modelo del Dominio y tienen que ser identificadas y diseñadas cuidadosamente.

**Conectores:**

**Modelo de Datos**

Este concepto existe a veces en la implementación de la Capa de Persistencia para poder definir e incluso visualizar gráficamente el modelo de datos „entidad-relación‟ de la aplicación. Este concepto suele ser proporcionado completamente por la tecnología O/RM concreta que se utilice, por lo que está completamente ligado a una infraestructura/tecnología específica (p.e. Entity Framework proporciona una forma de definir un modelo entidad-relación o incluso de exportarlo a partir de una base de datos existente).

**Tecnología de Persistencia (O/RM, etc.)**

Es simplemente la capa de infraestructura/tecnología utilizada internamente por nuestros Repositorios. Normalmente será, por lo tanto, la propia tecnología que hayamos escogido, bien un O/RM como Entity Framework, o simplemente tecnología de más bajo nivel como ADO.NET. Pero en este caso estaríamos hablando ya de tecnología, por lo que esto se explica en detalle en el subcapítulo de „Implementación de Capa de Persistencia‟, en la última parte del presente capítulo.

**Agentes de Servicios Distribuidos externos**

Cuando un componente del dominio debe acceder a datos proporcionados por un servicio distribuido externo (p.e. un Servicio-Web), debemos implementar código que gestione la semántica de comunicación con dicho servicio en particular. Estos Agentes de Servicio implementan precisamente componentes de acceso a datos que encapsulan y aíslan los requerimientos de los Servicios distribuidos e incluso pueden soportar aspectos adicionales como cache, soporte off-line y mapeos básicos entre el formato de datos expuesto en los Servicios distribuidos externos y el formato de datos requerido/utilizado por nuestra aplicación.