PROCESO DIRECCIÓN DE FORMACIÓN PROFESIONAL INTEGRAL

FORMATO GUÍA DE APRENDIZAJE

IDENTIFICACIÓN DE LA GUIA DE APRENDIZAJE

- Denominación del Programa de Formación: ANALISIS Y DESARROLLO DE SISTEMAS DE INFORMACION.
- Código del Programa de Formación: 228106
- Nombre del Proyecto:
- Fase del Proyecto: PLANEACIÓN
- Actividad de Proyecto: DISEÑAR LA ESTRUCTURA TECNOLÓGICA DEL SISTEMA INTEGRAL
- Competencia: PARTICIPAR EN EL PROCESO DE NEGOCIACIÓN DE TECNOLOGÍA INFORMÁTICA PARA PERMITIR LA IMPLEMENTACIÓN DEL SISTEMA DE INFORMACIÓN.
- Resultados de Aprendizaje: DEFINIR ESTRATEGIAS PARA LA ELABORACIÓN DE TÉRMINOS DE REFERENCIA Y PROCESOS DE EVALUACIÓN DE PROVEEDORES, EN LA ADQUISICIÓN DE TECNOLOGÍA, SEGÚN PROTOCOLOS ESTABLECIDOS.
- Duración de la Guía

2. PRESENTACIÓN

Para muchos, dominar la arquitectura de software es uno de los objetivos que han buscado durante algún tiempo, sin embargo, no siempre es claro el camino para dominarla, ya que la arquitectura de software es un concepto abstracto que cada persona lo interpreta de una forma diferente, dificultando con ello comprender y diseñar con éxito una arquitectura de software.

3. FORMULACIÓN DE LAS ACTIVIDADES DE APRENDIZAJE

- Materiales:
  Git, Visual code
  Portatil ó Computador de Escritorio
- Ambiente Requerido
- Descripción de la(s) Actividad(es)

EVOLUCIÓN DEL DESARROLLO DE SOFTWARE

En la imagen 1 se visualizan los grandes hitos de la evolución del desarrollo de software, las necesidades emergentes de cada uno de esos momentos y los avances tecnológicos que surgieron como solución. Se pone énfasis en la necesidad presente desde el inicio de contar con estrategias y técnicas que permitieran el desarrollo de productos confiables en el marco de proyectos predecibles. Esta necesidad fue la que motivó la creación de la Ingeniería de Software

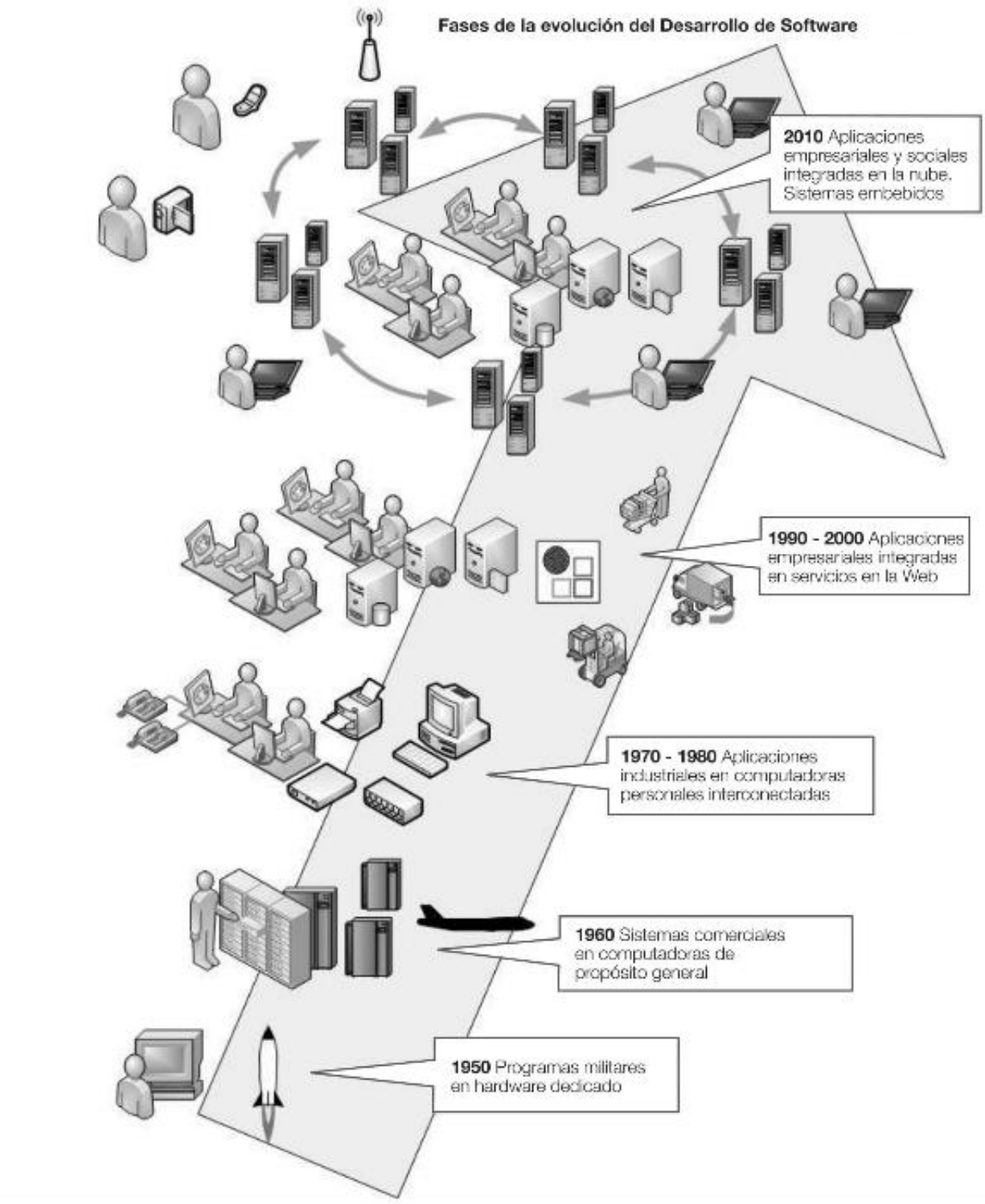

Imagen 1 – Evolución del desarrollo de software

ARQUITECTURA DE SOFTWARE

Antes de comenzar revisemos algunas definiciones de arquitectura de software:

“La arquitectura es un nivel de diseño que hace foco en aspectos "más allá de los algoritmos y estructuras de datos de la computación; el diseño y especificación de la estructura global del sistema es un nuevo tipo de problema “

— "An introduction to Software Architecture" de David Garlan y Mary Shaw

“La Arquitectura de Software se refiere a las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos. “

— Software Engineering Institute (SEI)

“El conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de software, relaciones entre ellos, y las propiedades de ambos. “

— Documenting Software Architectures: Views and Beyond (2nd Edition), Clements et al, AddisonWesley, 2010

“La arquitectura de software de un programa o sistema informático es la estructura o estructuras del sistema, que comprenden elementos de software, las propiedades visibles externamente de esos elementos y las relaciones entre ellos. “

— Software Architecture in Practice (2nd edition), Bass, Clements, Kazman; AddisonWesley 2003

¿Que tienen en común las anteriores definiciones?

Todas coinciden en que la arquitectura **se centra en la estructura del sistema, los componentes que lo conforman y la relación que existe entre ellos**.

PATRONES DE DISEÑO

Los patrones de diseño tienen un impacto relativo con respecto a un componente, esto quiere decir que tiene un impacto menor sobre todo el componente. Dicho de otra forma, si quisiéramos quitar o remplazar el patrón de diseño, solo afectaría a las clases que están directamente relacionadas con él, y un impacto imperceptible para el resto de componentes que conforman la arquitectura.

A principios de la década de 1990 fue cuando los patrones de diseño tuvieron su gran debut en el mundo de la informática a partir de la publicación del libro Design Patterns, escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes que ya se utilizaban sin ser reconocidos como patrones de diseño.

Es importante mencionar que la utilización de patrones de diseño demuestra la madurez de un programador de software ya que utiliza soluciones probadas para problemas concretos que ya han sido probados en el pasado. Toma en cuenta que el dominio de los patrones de diseño es una práctica que se tiene que

perfeccionar y practicar, es necesario conocer las ventajas y desventajas que ofrece cada uno de ellos, pero sobre todo requiere de experiencia para identificar dónde se deben de utilizar.

Lo más importante de utilizar los patrones de diseño es que evita tener que reinventar la rueda, ya que son escenarios identificados y su solución está documentada y probada por lo que no es necesario comprobar su efectividad. Los patrones de diseño se basan en las mejores prácticas de programación.

Los patrones de diseño pretenden:

 Proporcionar un catálogo de soluciones probadas de diseño para problemas comunes conocidos.

 Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.

 Crear un lenguaje estándar entre los desarrolladores. Facilitar el aprendizaje a nuevas generaciones de programadores.

Asimismo, no pretenden:

 Imponer ciertas alternativas de diseño frente a otras.

 Imponer la solución definitiva a un problema de diseño.

 Eliminar la creatividad inherente al proceso de diseño.

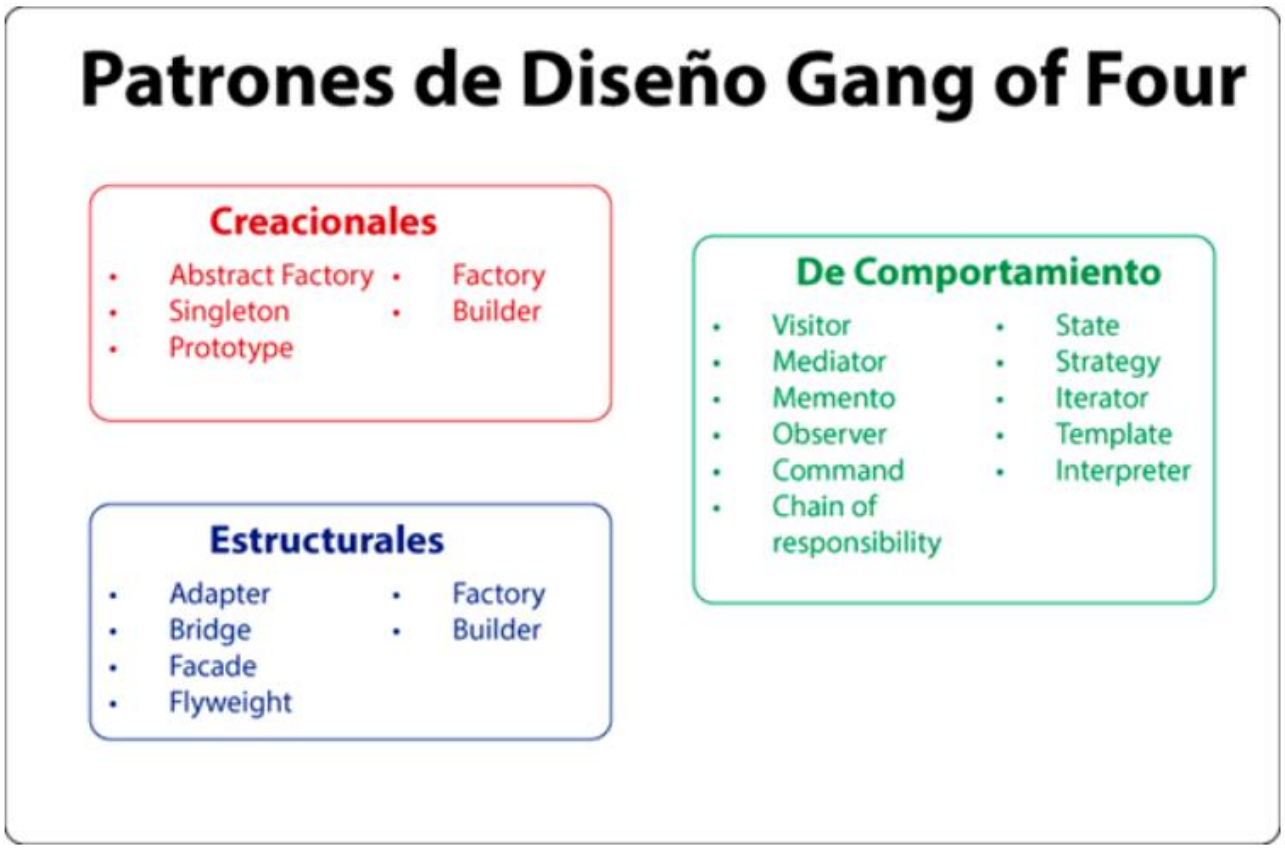

Imagen 2 – Lista de los patrones de diseño

PATRONES ARQUITECTONICOS

Los patrones arquitectónicos tienen un gran impacto sobre el componente, lo que quiere decir que cualquier cambio que se realice una vez construido el componente podría tener un impacto mayor.

A continuación, veremos las principales definiciones de los patrones arquitectónicos.

“Expresa una organización estructural fundamental o esquema para sistemas de software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y pautas para organizar las relaciones entre ellos. “

— TOGAF

“Los patrones de una arquitectura expresan una organización estructural fundamental o esquema para sistemas complejos. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades únicas e incluye las reglas y pautas que permiten la toma de decisiones para organizar las relaciones entre ellos. El patrón de arquitectura para un sistema de software ilustra la estructura de nivel macro para toda la solución de software. Un patrón arquitectónico es un conjunto de principios y un patrón de grano grueso que proporciona un marco abstracto para una familia de sistemas. Un patrón arquitectónico mejora la partición y promueve la reutilización del diseño al proporcionar soluciones a problemas recurrentes con frecuencia. Precisamente hablando, un patrón arquitectónico comprende un conjunto de principios que dan forma a una aplicación.”

— Achitectural Patterns - Pethuru Raj, Anupama Raman, Harihara Subramanian

“Los patrones de arquitectura ayudan a definir las características básicas y el comportamiento de una aplicación.”

— Software Architecture Patterns - Mark Richards

Tanto los patrones de diseño cómo los patrones arquitectónicos pueden resultar similares ante un arquitecto inexperto, pero por ninguna razón debemos confundirlos, ya son cosas diferentes.

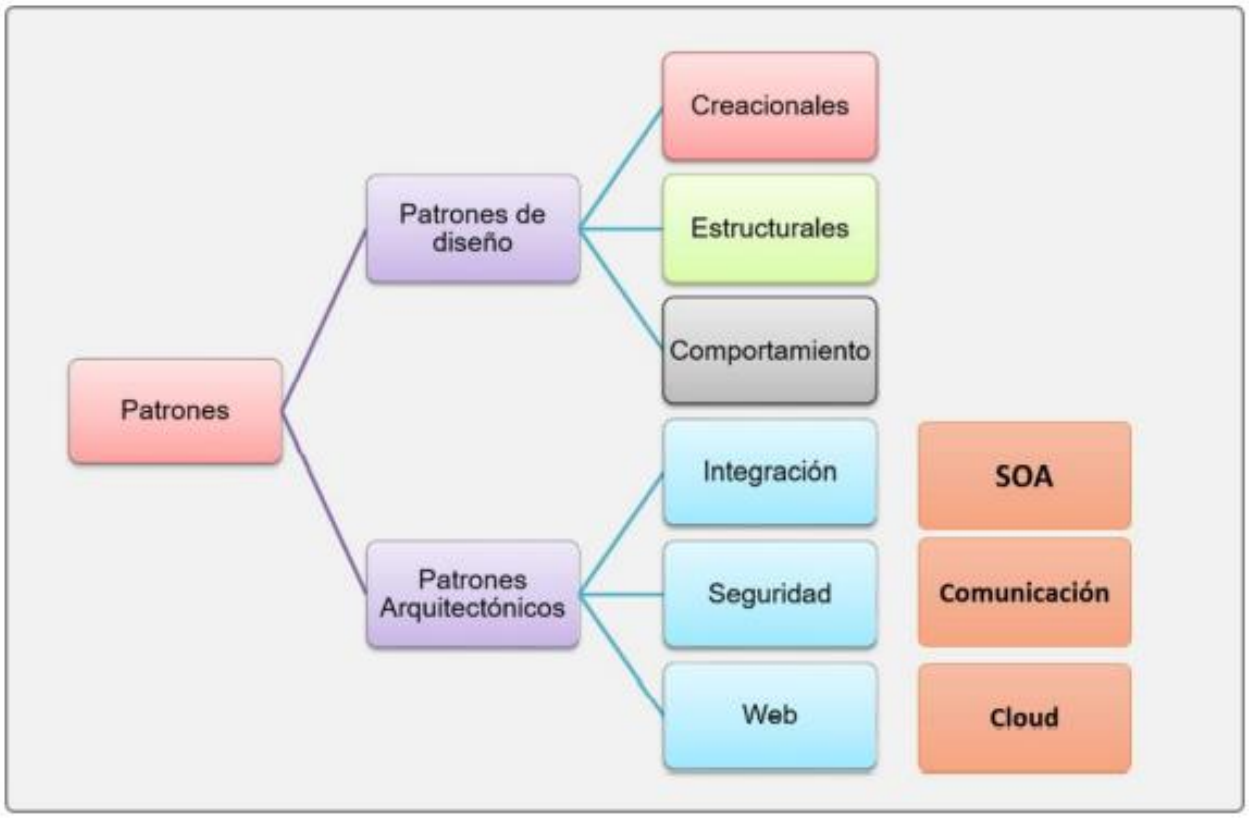

Imagen 3 – Consolidado de patrones

¿CÓMO DIFERENCIAR UN PATRÓN ARQUITECTÓNICO?

Los patrones arquitectónicos son fáciles de reconocer debido a que tiene un impacto global sobre la aplicación, e incluso, el patrón rige la forma de trabajar o comunicarse con otros componentes, es por ello que a cualquier cambio que se realice sobre ellos tendrá un impacto directo sobre el componente, e incluso, podría tener afectaciones con los componentes relacionados.

Un ejemplo típico de un patrón arquitectónico es el denominado “Arquitectura en 3 capas”, el cual consiste en separar la aplicación 3 capas diferentes, las cuales corresponden a la capa de presentación, capa de negocio y capa de datos:

Imagen 4 – Arquitectura de capas

ESTILOS ARQUITECTÓNICOS

Un estilo arquitectónico, es necesario regresarnos un poco a la arquitectura tradicional (construcción), para ellos, un estilo arquitectónico es un método específico de construcción, caracterizado por las características que lo hacen notable y se distingue por las características que hacen que un edificio u otra estructura sea notable o históricamente identificable.

En el software aplica exactamente igual, pues un estilo arquitectónico determina las características que debe tener un componente que utilice ese estilo, lo cual hace que sea fácilmente reconocible. De la misma forma que podemos determinar a qué periodo de la historia pertenece una construcción al observar sus características físicas, materiales o método de construcción, en el software podemos determinar que estilo de arquitectura sigue un componente al observar sus características.

Algunas definiciones de estilos arquitectónicos:

"Un estilo arquitectónico define una familia de sistemas en términos de un patrón de organización estructural; Un vocabulario de componentes y conectores, con restricciones sobre cómo se pueden combinar. "

— M. Shaw and D. Garlan, Software architecture: perspectives on an emerging discipline. Prentice Hall, 1996.

"Un estilo de arquitectura es una asignación de elementos y relaciones entre ellos, junto con un conjunto de reglas y restricciones sobre la forma de usarlos"

—Clements et al., 2003

"Un estilo arquitectónico es una colección con nombre de decisiones de diseño arquitectónico que son aplicables en un contexto de desarrollo dado, restringen decisiones de diseño arquitectónico que son específicas de un sistema particular dentro de ese contexto, y obtienen cualidades beneficiosas en cada uno sistema resultante."

—R. N. Taylor, N. Medvidović and E. M. Dashofy, Software architecture: Foundations, Theory and Practice. Wiley, 2009

LA RELACIÓN ENTRE PATRONES DE DISEÑO, ARQUITECTÓNICOS Y ESTILOS ARQUITECTÓNICOS

Para una gran mayoría de los arquitectos, incluso experimentados, les es complicado diferenciar con exactitud que es un patrón de diseño, un patrón arquitectónico y un estilo arquitectónico, debido principalmente a que, como ya vimos, no existe definiciones concretas de cada uno, además, no existe una línea muy delgada que separa a estos tres conceptos.

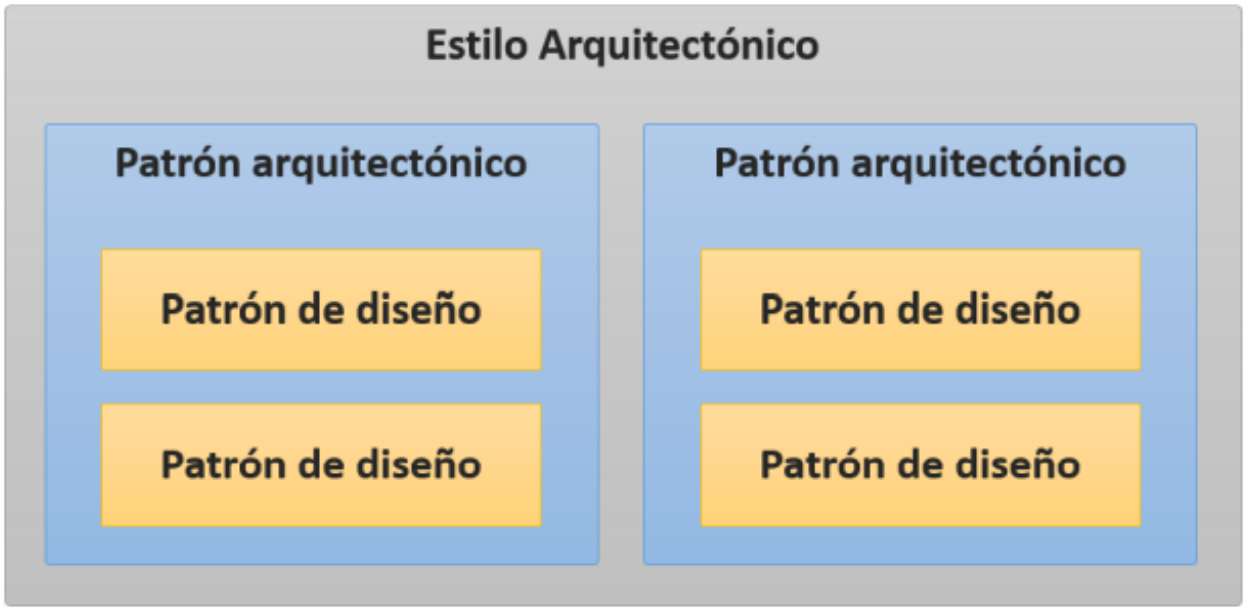

Imagen 5 – Relación entre estilo arquitectónico, patrón arquitectónico y patrón de diseño

ESTILOS ARQUITETONICOS

Un estilo arquitectónico establece un marco de referencia a partir del cual es posible construir aplicaciones que comparten un conjunto de atributos y características mediante el cual es posible identificarlos y clasificarlos.

Representational State Transfer (REST)

REST se ha convertido sin lugar a duda en uno de los estilos arquitectónicos más utilizados en la industria, ya que es común ver que casi todas las aplicaciones tienen un API REST que nos permite interactuar con ellas por medio de servicios.

Lo primero que tenemos que saber es que REST es un conjunto de restricciones que crean un estilo arquitectónico y que es común utilizarse para crear aplicaciones distribuidas. REST fue nombrado por primera vez por Roy Fielding en el año 2000 donde definió a REST como:

REST proporciona un conjunto de restricciones arquitectónicas que, cuando se aplican como un todo, enfatizan la escalabilidad de las interacciones de los componentes, la generalidad de las interfaces, la implementación independiente de los componentes y los componentes intermedios para reducir la latencia de interacción, reforzar la seguridad y encapsular los sistemas heredados. — Roy Fielding

Como se estructura REST

REST describe 3 conceptos clave, que son: Datos, Conectores y Componentes, los cuales trataremos de definir a continuación.

Datos

Uno de los aspectos más importantes que propone REST es que los datos deben de ser transmitidos a donde serán procesados, en lugar de que los datos sean transmitidos ya procesados, pero ¿qué quiere decir esto exactamente?

En cualquier arquitectura distribuida, son los componentes de negocio quien procesa la información y nos responde la información que se produjo de tal procesamiento, por ejemplo, una aplicación web tradicional, donde la vista es construida en el backend y nos responde con la página web ya creada, con los elementos HTML que la componente y los datos incrustados. En este sentido, la aplicación web proceso los datos en el backend, y como resultado nos arroja el resultado de dicho procesamiento, sin embargo, REST lo que propone es que los datos sean enviados en bruto para que sea el interesado de los datos el que los procese para generar la página (o cualquier otra cosa que tenga que generar), en este sentido, lo que REST propone no es generar la página web, si no que mandar los datos en bruto para que el consumidor sea el responsable de generar la página a través de los datos que responde REST.

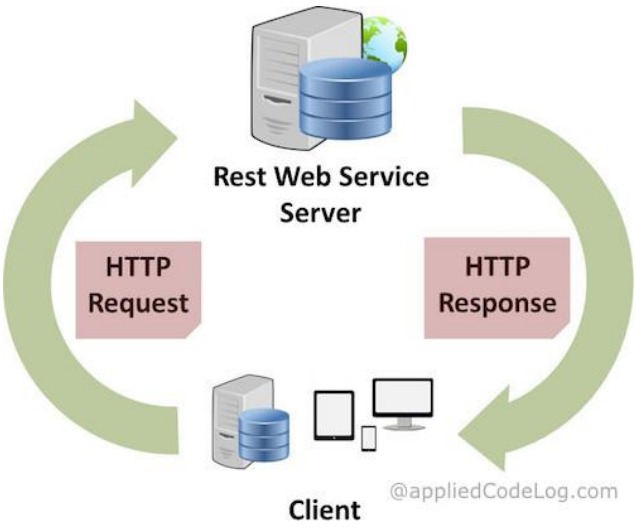

Imagen 6 – Procesamiento

En la imagen anterior podemos apreciar más claramente lo que comentábamos anteriormente, en el cual, los datos son procesados por el servidor y como resultado se obtiene un documento HTML procesado con todos los datos a mostrar.

Este enfoque fue utilizado durante mucho tiempo, pues las aplicaciones web requerían ir al servidor a consultar la siguiente página y este les regresa el HTML que el navegador solamente tenía que renderizar, sin embargo, que pasa con nuevas tecnologías como Angular, Vue, o React que se caracterizan por generar la siguiente página directamente en el navegador, lo que significa que no

requieren ir al servidor por la siguiente página y en su lugar, solo requieren los datos del servidor presentarlo al usuario.

Bajo este enfoque podemos entender que REST propone servir los datos en crudo para que sea el consumidor quien se encargue de procesar los datos y mostrarlo como mejor le convenga.

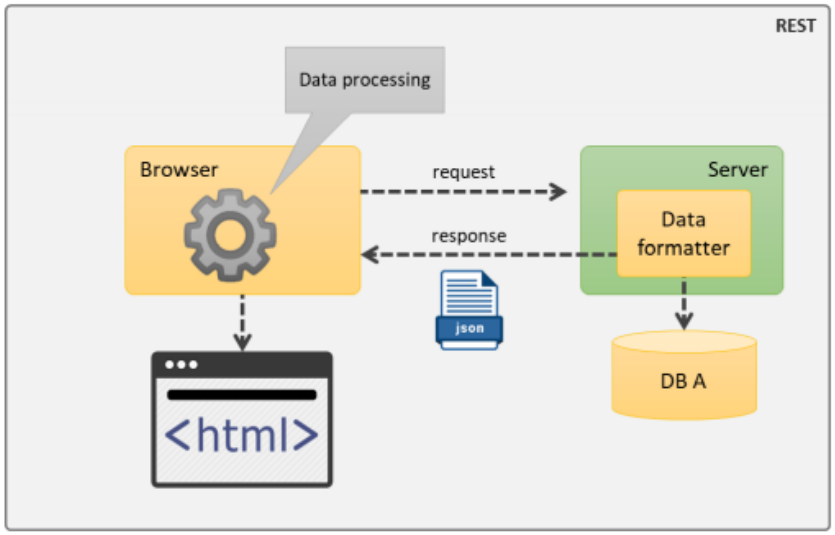

Imagen 7 – Procesamiento del lado del cliente

La imagen anterior representa mejor lo que acabamos de decir, pues ya vemos que el servidor solo se limita a servir la información en un formato esperado por el cliente, el cual por lo general es JSON, aunque podría existen más formatos. Por otra parte, el cliente recibe esa información, la procesa y genera la vista a partir de los datos proporcionados por el servidor, de esta forma, varias aplicaciones ya sean Web, Móviles, escritorio, o en la nube pueden solicitar esa misma información y mostrarla de diferentes formas, sin estar limitados por el formato procesado que regresa el servidor, como sería el caso de una página HTML.

Ahora bien, el caso de la página HTML es solo un ejemplo, pero se podría aplicar para cualquier otra cosa que no sea una página, como mandar los datos a una app móvil, imágenes o enviarlos a otro servicio para su procesamiento, etc.

Recursos

En REST se utiliza la palabra Recurso para hacer referencia a la información, y de esta forma, todos lo que podamos nombrar puede ser un recurso, como un documento, imagen, servicios, una colección de recursos, etc. En otras palabras, un recurso puede ser cualquier cosa que pueda ser recuperada por una referencia de hipertexto (URL)

Cada recurso tiene una dirección única que permite hacerle referencia, de tal forma que ningún otro recurso puede compartir la misma referencia, algo muy parecido con la web, donde cada página tiene un dominio y no pueden existir dos páginas web en el mismo dominio, o dos imágenes en la

misma URL, de la misma forma, en REST cada recurso tiene su propia URL o referencia que hace posible recuperar un recurso de forma inconfundible.

Representaciones

Una de las características de los recursos es que pueden tener diferentes representaciones, es decir, que un mismo recurso puede tener diferentes formatos, por ejemplo, una imagen que está representada por un recurso (URL) puede estar en formato PNG o JPG en la misma dirección, con la única diferencia de que los metadatos que describen el recurso cambian según la representación del recurso.

En este sentido, podemos decir que una representación consta de datos y metadatos que describen los datos. Los metadatos tienen una estructura de nombre-valor, donde el nombre describe el metadato y el valor corresponde al valor asignado al metadato. Por ejemplo, un metadato para representar una representación en formato JSON sería “Content-Type”=”application/json” y el mismo recurso en formato XML sería “Content-Type”=”application/xml”.

Conectores

Los conectores son los componentes que encapsulan la actividad de acceso a los recursos y transferencia, de esta forma, REST describe los conectores como interfaces abstractas que permite la comunicación entre compontes, mejorando la simplicidad al proporcionar una separación clara de las preocupaciones y ocultando la implementación subyacente de los recursos y los mecanismos de comunicación.

Por otro lado, REST describe que todas las solicitudes a los conectores sean sin estado, es decir, que deberán contener toda la información necesaria para que el conector comprenda la solicitud sin importar las llamadas que se hallan echo en el pasado, de esta forma, ante una misma solicitud deberíamos de obtener la misma respuesta. Esta restricción cumple cuatro funciones:

1. Eliminar cualquier necesidad de que los conectores retengan el estado de la aplicación entre solicitudes, lo que reduce el consumo de recursos físicos y mejora la escalabilidad.

2. Permitir que las interacciones se procesen en paralelo sin requerir que el mecanismo de procesamiento entienda la semántica de la interacción.

3. Permite a un intermediario ver y comprender una solicitud de forma aislada, lo que puede ser necesario cuando los servicios se reorganizan dinámicamente.

4. Facilita el almacenamiento en caché, lo que ayuda a reutilizar respuestas anteriores.

En REST existen dos tipos de conectores principales, el cliente y el servidor, donde el cliente es quien inicial la comunicación enviando una solicitud al servidor, mientras que el servidor es el que escucha las peticiones de los clientes y responde las solicitudes para dar acceso a sus servicios.

Otro tipo de conector es el conector de caché, el cual puede estar ubicado en la interfaz de un conector cliente o servidor, el cual tiene como propósito almacenar las respuestas y reutilizarlas en

llamas iguales en el futuro, evitando con ello, solicitar recursos consultados anteriormente al servidor. El cliente puede evitar hacer llamadas adicionales al servidor, mientras que el servidor puede evitar sobrecargar a la base de datos, disco o procesador con consultas repetitivas. En ambos casos se mejora la latencia.

Componentes

Los componentes con software concretos que utilizan los conectores para consultar los recursos o servirlos, ya sea una aplicación cliente como lo son los navegadores (Chrome, Firefox, etc) o Web Servers como Apache, IIS, etc. Pero también existen los componentes intermedios, que actúan como cliente y servidor al mismo tiempo, como es el caso de un proxy o túnel de red, los cuales actúan como servidor al aceptar solicitudes, pero también como cliente a redireccionar las peticiones a otro servidor.

Los componentes se pueden confundir con los conectores, sin embargo, un conector es una interface abstracta que describe la forma en que se deben de comunicar los componentes, mientras que un componente es una pieza de software concreta que utiliza los conectores para establecer la comunicación.

Características de REST

Analizaremos las características que distinguen al estilo REST del resto. Las características no son ventajas o desventajas, si no que más bien, nos sirven para identificar cuando una aplicación sigue un determinado estilo arquitectónico.

Las características se pueden convertir en ventajas o desventajas solo cuando analizamos la problemática que vamos a resolver, mientras tanto, son solo características:

1. Todos los recursos deben de poder ser nombrados, es decir, que tiene una dirección (URL) única que la diferencia de los demás recursos.

2. Un recurso puede tener más de una representación, donde un recurso puede estar representado en diferentes formatos al mismo tiempo y en la misma dirección.

3. Los datos están acompaños de metadatos que describen el contenido de los datos.

4. Las solicitudes tienen toda la información para poder ser entendidas por el servidor, lo que implica que podrán ser atendidas sin importar las invocaciones pasadas (sin estado) ni el contexto de la misma.

5. Debido a la naturaleza sin estado de REST, es posible cachear las peticiones, de tal forma que podemos retornar un resultado previamente consultado sin necesidad de llamar al servidor o la base de datos.

6. REST es interoperable, lo que implica que no está casado con un lenguaje de programación o tecnología especifica.

RESTful y su relación con REST

Sin lugar a duda, uno de los conceptos que más se confunden cuando hablamos de arquitecturas REST, es el concepto de RESTfull, pues aún que podrían resultar similares, tiene diferencias fundamentales que las hacen distintas.

Como ya habíamos mencionado antes, REST parte de la idea de que tenemos datos que pueden tener diferentes representaciones y los datos siempre están acompañados de metadatos que describen el contenido de los datos, además, existen conectores que permiten que se pueda establecer una conexión, los cuales son abstractos desde su definición, finalmente, existen los componentes que son piezas de software concretas que utilizan los conectores para transmitir los datos.

Por otra parte, RESTfull toma las bases de REST para crear servicios web mediante el protocolo HTTP, de esta forma, podemos ver que los conectores ya no son abstractos, si no que delimita que la comunicación siempre tendrá que ser por medio de HTTP, además, define que la forma para recuperar los datos es por medio de los diferentes métodos que ofrece HTTP, como lo son GET, POST, DELETE, PUT, PACHT. En este sentido, podemos observar que REST jamás define que protocolo utilizar, y mucho menos que debemos utilizar los métodos de HTTP para realizar las diferentes operaciones.

Entonces, podemos decir que RESTful promueve la creación de servicios basados en las restricciones de REST, utilizando como protocolo de comunicación HTTP.

También podemos observar que los servicios RESTful mantiene las propiedades de los datos que define REST, ya que en RESTful, todos los recursos deben de poder ser nombrados, es decir, deberán tener un URL única que pueda ser alcanzada por HTTP, por otra parte, define una serie de metadatos que describen el contenido de los datos, es por ello que con RESTful podemos retornar un XML, JSON, Imagen, CSS, HTML, etc. Cualquier cosa que pueda ser descrita por un metadato, puede ser retornada por REST.

Ventajas

• Interoperable: REST no está casado con una tecnología, por lo que es posible implementarla en cualquier lenguaje de programación.

• Escalabilidad: Debido a todas las peticiones son sin estado, es posible agregar nuevos nodos para distribuir la carga entre varios servidores.

• Reducción del acoplamiento: Debido a que los conectores definen interfaces abstractas y que los recursos son accedidos por una URL, es posible mantener un bajo acoplamiento entre los componentes.

• Testabilidad: Debido a que todas las peticiones tienen toda la información y que son sin estado, es posible probar los recursos de forma individual y probar su funcionalidad por separado.

• Reutilización: Una de los principios de REST es llevar los datos sin procesar al cliente, para que sea el cliente quién decida cuál es la mejor forma de representar los datos al usuario, por este motivo, los recursos de REST pueden ser reutilizado por otros componentes y representar los datos de diferentes maneras.

Desventajas

• Performance: Como en todas las arquitecturas distribuidas, el costo en el performance es una constante, ya que cada solicitud implica costos en latencia, conexión y autenticaciones.

• Más puntos de falla: Nuevamente, las arquitecturas distribuidas traen con sigo ciertas problemáticas, como es el caso de los puntos de falla, ya que al haber múltiples componentes que conforman la arquitectura, multiplica los puntos de falla.

• Gobierno: Hablando exclusivamente desde el punto de vista de servicios basados en REST, es necesario mantener un gobierno sobre los servicios implementados para llevar un orden, ya que es común que diferentes equipos creen servicios similares, provocando funcionalidad repetida que tiene que administrarse.

Cuando debo de utilizar el estilo REST

Actualmente REST es la base sobre la que están construidas casi todas las API de las principales empresas del mundo, con Google, Slack, Twitter, Facebook, Amazon, Apple, etc, incluso algunas otras como Paypal han migrado de los Webservices tradicionales (SOAP) para migrar a API REST.

En este sentido, REST puede ser utilizado para crear integraciones con diferentes dispositivos, incluso con empresas o proveedores externos, que requieren información de nuestros sistemas para opera con nosotros. Por ejemplo, mediante API REST, Twitter logra que existan aplicaciones que puedan administrar nuestra cuenta, pueda mandar Tweets, puedan seguir personas, puedan retwittear y todo esto sin entrar a Twitter. Otro ejemplo, es Slack, que nos permite mandar mensaje a canales, enviar notificaciones a un grupo si algo pasa en una aplicación externa, como Trello o Jira.

En Google Maps, podemos ir guardando nuestra posición cada X tiempo, podemos analizar rutas, analizar el tráfico, tiempo de llega a un destino, todo esto mediante el API REST, y sin necesidad de entrar a Google Maps para consultar la información.

En otras palabras, REST se ha convertido prácticamente en la Arquitectura por defecto para integrar aplicaciones, incluso ha desplazado casi en su totalidad a los Webservices tradicionales (SOAP).

Entonces, cuando debo de utilizar REST, la respuesta es fácil, cuando necesitamos crear un API que nos permite interactuar con otra plataformas, tecnologías o aplicaciones, sin importar la tecnología y que necesitemos tener múltiples representaciones de los datos, como consultar información en formato JSON, consultar imágenes en PNG o JPG, envías formularios directamente desde un formulario HTML o simplemente enviar información binaria de un punto a otro, ya que

REST permite describir el formato de la información mediante metadatos y no está casado un solo formato como SOAP, el cual solo puede enviar mensajes en XML.

EJERCICIO

Suponer que el backend está en un solo servidor.

1. ¿Como realizo escalamiento vertical de una aplicación basada en REST? ¿Que necesito para poderlo realizar?
2. ¿Como realizo escalamiento horizontal de una aplicación basada en REST? ¿Que necesito para poderlo realizar?
3. ¿Puedo utilizar un load balancer en una aplicación basada en REST? Defina su respuesta.
4. ¿Puedo implementar algún estilo arquitectónico visto con REST? ¿Cual y por qué?
5. Como puedo utilizar todos los endpoint vistos en clase en un mismo servidor para que el backend tenga alta disponibilidad - Recordar que la disponibilidad es un atributo de calidad. (incluye explicación de la fórmula)
6. ¿Cuál sería la diferencia entre un servicio web SOAP y RESTful?
7. ¿Como puedo consumir un servicio RESTfull en javascript, angular, vue o react?
8. Como puedo consumir un servicio RESTfull en Android, Swift, react native, ionic o flutter
9. Realice el consumo del endpoint de cada grupo con los 4 métodos vistos en clase, realice una descripción del resultado de cada uno.

CASO DE ESTUDIO

Realizar el caso de estudio anexo donde se evidencie el diagrama de componentes, diagrama de despliegue, creado para solucionar la problemática planteada y que permita gestionar el proceso de negocio en una arquitectura basada en REST. Adicional cree la estructura del proyecto en github.

Suponga que cada servicio que expone es un endpoint que se publica en los puertos estipulados y para gestionar el proceso de negocio se exponen 4 endpoint.

4. ACTIVIDADES DE EVALUACIÓN

Evidencia de Conocimiento: Realizar un mapa conceptual digital sobre la arquitectura REST, realice una presentación sobre las preguntas, los diagramas del caso de estudio y el proyecto en github.

Evidencia de Desempeño: Realiza la sustentación de cada evidencia de conocimiento.

Evidencia de Producto: Entrega de los documentos requeridos en las evidencias de conocimiento.

Evidencias de Aprendizaje	Criterios de Evaluación	Técnicas e Instrumentos de Evaluación
Evidencias de Conocimiento:	**Reconoce las principales características de la arquitectura monolítica**	Redacción
Evidencias de Desempeño:	**Interpreta el uso de una aplicación monolítica en un entorno empresarial**	Presentación
Evidencias de Producto	**Realiza el entregable con la documentación completa**	Entrega de la guía con todas las evidencias requeridas.

1. GLOSARIO DE TÉRMINOS

De acuerdo a la práctica realizar su propio glosario de términos.

6. REFERENTES BIBLIOGRÁFICOS

Construya o cite documentos de apoyo para el desarrollo de la guía, según lo establecido en la guía de desarrollo curricular

7. CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	**Néstor Rodríguez**	**Instructor**	**Teleinformática**	**NOVIEMBRE-2020**

8. CONTROL DE CAMBIOS (diligenciar únicamente si realiza ajustes a la guía)

	Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
Autor (es)					