Programación en Ambiente Web 2020

Segunda evaluación parcial

Salinas Leonardo - 104478 - Luján

1) ¿Qué diferencia existe entre una petición HTTP generada por un agente de usuario de forma asincrónica respecto a una sincrónica? ¿Cómo puede distinguir una aplicación web entre ambas?

El concepto de peticiones síncronas o asíncronas surgen con la programación AJAX.

Una petición HTTP generada de forma síncrona causa que la ejecución del código JavaScript del lado del cliente no pueda continuar hasta tanto no se reciba una respuesta por parte del servidor. Con lo cual, los sitios web que funcionan de esta forma no pueden generar múltiples consultas al servidor en simultáneo, sino que van a generarse de la forma petición-respuesta, una a una, y todas en el mismo orden. Desde el punto de vista de la experiencia del usuario, cuando una petición de este tipo es generada, el sitio queda bloqueado hasta que el servidor retorne una respuesta.

Las peticiones HTTP asíncronas, en cambio, son peticiones completamente independientes a las demás. Por lo tanto, múltiples peticiones y respuestas pueden estar circulando al mismo tiempo y, al no haber dependencias entre ellas, pueden ir y venir en distinto orden.

Por supuesto, las peticiones síncronas están totalmente desaconsejadas. De hecho, están en vías de dejar de pertenecer al estándar.

2) ¿Qué diferencias existen entre el diseño responsivo y el universal? ¿En qué conceptos hay que hacer hinca-pié al momento de definir las media querys en cada caso?

Ambas estrategias de diseño lo que intentan es lograr diseñar los sitios web de forma tal que la visualización del mismo se adapte al dispositivo sobre el cual se está visualizando. Por ejemplo: PCs, teléfonos móviles, tablets (en modo portrait o landscape en ambos casos), smartTVs, etc. Para esto, el *responsive web design* utiliza mediaQuerys las cuales son reglas CSS que se aplican cuando la resolución de la pantalla toma ciertos valores. Al pasar ese umbral, algunas reglas CSS se desactivan y otras se activan, logrando que la visualización del sitio cambie de manera favorable. Esto incluye agregar o eliminar elementos, modificar su tamaño, cambiar la estructura con la que se muestran los datos, ocultar cierta información con menos relevancia para dejar espacio a otra información más importante, etc. Por lo tanto, lo que se

generan son múltiples conjuntos de reglas específicas para la resolución que se tiene en ese momento, y al pasar cierto umbral, se pasa de un conjunto a otro.

El *diseño universal*, en cambio, pretende que el diseño del sitio web sea tan flexible como para que se siga viendo bien, aún con diferentes resoluciones de pantalla, sin tener que modificar las reglas del CSS. Esto se logra mediante el uso de modelos de diseño como Flexbox, que permiten que los componentes se muevan y desplacen dependiendo de la resolución, pero no intervienen de por medio otras reglas CSS.

Como nota al margen quiero aclarar que, buscando información, encontré otros nombres para ambas estrategias. A lo que yo menciono anteriormente como *diseño universal* es en realidad el *responsive web design*, mientras que a lo que yo llamé *responsive web design* es en realidad el *diseño adaptativo (adaptative web design)*. Digo esto para dejar en claro que, a pesar de esta diferencia de semántica, la diferencia entre ambas estrategias es bien clara: una modifica las reglas CSS a medida que la resolución cambia, mientras que la otra no lo hace.

3) ¿Por qué decimos que no son directamente comparables REST y SOAP en el contexto de los Web Services?

La respuesta corta sería que REST y SOAP no son directamente comparables porque esencialmente no son lo mismo. SOAP es un protocolo orientado al servicio y a la comunicación, mientras que REST es una definición de una arquitectura orientada al recurso.

Siendo más específicos sobre las diferencias de cada uno, para no extenderme demasiado, mencionaré brevemente algunas características en las cuales difieren:

REST es mucho más simple y con mejor performance (utiliza menos ancho de banda). Tiene mejor soporte para el lenguaje JavaScript y, si bien se habla mucho sobre que SOAP es más seguro que REST, lo cierto es que ambos pueden ser seguros o inseguros dependiendo de la implementación. Lo que sí, como SOAP está mucho mejor documentado y es más robusto (utiliza WSDL como definición, posee un error-handler, etc.), es más difícil que los errores pasen desapercibidos.

Por todo esto, como REST es más versátil, simple y eficiente, es mucho más usado que SOAP y tiende a serlo cada vez más.

4) Explique brevemente tres principios de desarrollo seguro y de un ejemplo para cada uno.

Si bien el libro *"Secure Coding Principles and Practices"* de Graff y van Wyk nombraba varios principios de desarrollo seguro, voy a nombrar tres que me parecieron particularmente interesantes:

El principio del menor privilegio. Darle a un usuario los permisos mínimos que necesita para el trabajo que debe hacer. Ni más (para no aumentar el riesgo) ni menos (para que pueda efectivamente hacer su trabajo). Los privilegios son otorgados cuando se necesitan y son removidos cuando ya no lo son. Por ejemplo, que un gerente de la sección de ventas no pueda tener permisos de modificación en archivos del sector de producción.

Reusar código conocido como seguro: La idea es simple. Un código que es conocido por cumplir con la función que se necesita y que lo hace de forma segura, es mucho más seguro que programar esa funcionalidad desde cero. Un ejemplo de esto es utilizar algoritmos de encriptación conocidos como seguros. Tratar de innovar y crear un algoritmo de encriptación propio definitivamente no va a tener la misma confiabilidad que uno que es mundialmente usado y conocido por ser seguro. En este caso no sólo se gana en seguridad, sino también en tiempo y costo.

Seguridad en cada capa: La seguridad debe implementarse durante todo el ciclo de vida del sistema y en cada capa. Por ejemplo, configurar correctamente el firewall, pero también verificar que los sistemas críticos tengan fuentes redundantes de alimentación, y que también el frontend verifique que el usuario no está ingresando caracteres especiales en un campo donde no debería haberlos, pero obviamente, aún si el usuario en el frontend logra enviar datos sospechosos, que el backend pueda detectar esto y que rechace la petición.

5) ¿Cómo se relaciona el header HTTP Content-Security-Policy con la seguridad de un sistema web y por qué es fundamental su uso hoy en día? ¿Se puede implementar esto mismo de otra forma que no sea vía header HTTP (a nivel del server web)?

El header HTTP Content-Security-Policy permite controlar qué recursos pueden ser cargados por el browser. Especifica una "White-list" de servidores "confiables". Los recursos provenientes de estos servidores, ya sean scripts, CSS, etc., serán ejecutados por el browser y, cualquier otro recurso proveniente de un servidor desconocido será ignorado y el browser no lo ejecutará.

El uso de este header es usado principalmente para evitar el cross-site scripting (XSS) el cual consiste en hacer que un user-agent ejecute código malicioso de terceros.

Una alternativa a este header sería utilizando la etiqueta <meta> en el head del documento HTML. De la siguiente forma:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://*;">
```

En este caso, para una imagen.

6) ¿Por qué es útil un buen análisis de riesgos a la hora de priorizar las mejoras de seguridad que podamos aplicar a nuestro sistema web?

El análisis de riesgo es fundamental a la hora de priorizar las mejores de seguridad ya que permiten identificar aquellos riesgos que son potencialmente más peligrosos para nuestro sistema. De esta forma se pueden armar estrategias para solventar esos riegos, comenzando con aquellos que puedan causar más problemas.

Un análisis de riesgos sencillo puede ser el de la probabilidad por el impacto. Se analiza cada funcionalidad del sistema con posibles vectores de ataques y se le da un puntaje en base a cuán probable es que ocurra un ataque, y cuanto impacto tendría en caso de que ocurra. En base al resultado del análisis se arma un gráfico que posiciona a la funcionalidad dependiendo de esos valores. Por ejemplo, una funcionalidad cuyo riesgo tiene poca probabilidad de ocurrencia y, si ocurre, cause poco impacto, no tendrá tanta prioridad. Pero un riesgo cuya probabilidad de ocurrencia sea medio, y de ocurrir, el impacto sea grande, sin dudas es mucho más prioritario el solucionarlo. De esta forma, se van posicionando en el gráfico las diferentes vulnerabilidades del sistema; y aquellas que requieran ser solucionadas con urgencia serán más fáciles de detectar. Un problema que puede ocurrir al efectuar éste análisis de riesgos es la dificultad de cuantificar las variables que entran en juego, sobre todo la probabilidad.

Un mal o nulo análisis de riesgos podría dejar brechas en el sistema por no darle prioridad a aquellos riesgos que podían causar mayores daños, creando así futuros vectores de ataques. Para efectuar un buen análisis de riesgos, hay muchas preguntas que deben responderse: ¿Quién usa la app? ¿Cuáles son los datos sensibles? Etc.

Si bien no es posible tener un sistema 100% seguro, lo que se busca es estar lo más cerca posible a ese valor.

7) Describa cómo generar una buena estrategia de SEO a partir del uso de herramientas semánticas.

Una buena estrategia de SEO debe incluir estrategias que permitan que el crawler de Google (por ejemplo) pueda recorrer nuestro sitio para así facilitarle la indexación. Para esto, se debe configurar correctamente el archivo "robots.txt" y diseñar nuestro sitio como el crawler espera. Existen diversas herramientas que permiten facilitar esta tarea. Para eso se pueden seguir estas estrategias:

- Palabras clave: Es esencial que nuestro sitio contenga las palabras clave más buscadas. Hay herramientas que permiten saber qué palabras fueron las que los usuarios buscaban al ingresar a nuestro sitio. En base a esto, aunque algunas palabras que puedan ser sinónimos, quizás una de ellas sea mucho más buscada que la otra.

- Respetar los estándares HTML5: Nuestro HTML debe respetar los estándares. Utilizar los tags que sean semánticamente correctos, colocar los atributos necesarios (como el atributo "alt" en las imágenes), etc.

- Utilizar el protocolo https: El paso de http a https es casi obligatorio para algunos browsers.

- Confeccionar correctamente el archivo "robots.txt" ya que es el que el crawler va a consumir.

- Confeccionar correctamente el sitemap del sitio para facilitarle al crawler la navegación por el mismo.

- Incorporar los tags <meta> correspondientes.

- Lograr que la página pueda generar texto enriquecido para que Google pueda mostrarlo en las búsquedas.

- Comprobar con qué otras entidades o palabras están relacionadas las palabras clave de tu sitio.

- Evitar penalizaciones: Algunas cosas en el diseño del sitio pueden causar penalizaciones por parte del crawler, causando el efecto opuesto al deseado. Por ejemplo, que nuestro sitio contenga las palabras clave más buscadas, pero que el contenido no esté relacionado con esas palabras.

- Hay muchos sitios web que ofrecen estas herramientas semánticas. Por ejemplo: https://technicalseo.com/tools/

8) ¿Cuáles son las ventajas y desventajas del modelo serverless en el cloud respecto al modelo tradicional basado en infraestructura (servers físicos / VMs).

El mantenimiento de un sitio mediante el modelo tradicional requiere que tanto el servidor como todos sus componentes (disco, memoria ram, procesadores, fuentes de alimentación, conexión a internet) corran por nuestra cuenta. Con lo cual el mantenimiento de dichos componentes también será nuestra responsabilidad.

En cambio, al contratar un servicio de cloud para alojar a nuestro sitio, los componentes antes mencionados correrán por parte del servicio que estemos contratando (Amazon, Azure, etc.).

Este modelo presenta varias ventajas y desventajas. Desde el punto de vista del servicio cloud, algunas ventajas serían:

- Además de los componentes de hardware necesarios, el mantenimiento y disponibilidad de toda la infraestructura será delegada al prestador del servicio (acondicionamiento del servidor, el lugar físico, fuentes redundantes, aires acondicionados, etc.)

- Es mucho más fácil de implementar ya que, dependiendo del tipo de servicio contratado, más o menos responsabilidades en cuanto a la configuración serán responsabilidad del prestador.

    - Iaas: Arquitectura orientada al servicio. Nuestra gestión comienza a partir del sistema operativo. Todo lo que está por debajo del sistema operativo (virtualización, servidores, almacenamiento y red) serán responsabilidad del prestador.

    - PaaS: El prestador del servicio nos brindará un entorno de desarrollo (ya sea para programar en Python, Java, etc. O para ejecutar nuestro código) sin importarnos la infraestructura (sistema operativo, etc.)

    - SaaS: Se le contrata al prestador únicamente el uso del software en específico (por ejemplo, correo electrónico, almacenamiento en la nube, etc.). El desarrollo del software, su mantenimiento y actualizaciones correrán por parte del prestador.

- Escalabilidad: Escalar en un modelo serverless es simple y menos costoso, ya que lo que debe hacerse es contratar más hardware y (en caso de ser necesario) migrar el sistema. Incluso el escalado puede ser por demanda: escalar hacia “arriba” cuando al carga es alta y hacia “abajo” cuando la carga es baja. Todo esto depende del servicio y las reglas establecidas. Características como el almacenamiento es virtualmente infinito.

- Pagar por lo que se usa: A diferencia de un modelo tradicional cuyo coste es fijo, en el servicio cloud se paga por lo que se usa (con mayor o menor granularidad dependiendo del prestador).

- Seguridad: Si se contrata a un prestador bueno, los datos alojados en un servicio de cloud casi siempre estarán más seguros que en nuestro propio datacenter.

- Accesibilidad: El prestador de servicio nos garantiza que nuestros datos podrán ser accesibles desde cualquier lugar, en cualquier momento. Incluso en distintas partes del planeta.

Si bien las ventajas pueden ser muchas más ya que hay una gran cantidad de servicios que se pueden contratar, creo que las ya mencionadas son suficientes.

Desventajas:

- Lock in: Cuando nuestro sistema ya está muy integrado al servidor del cloud, desligarse para migrar a otro cloud o a nuestro propio datacenter puede ser muy dificultoso dadas las dependencias que se generaron (aplicaciones contratadas, configuraciones realizadas, migración de los datos, etc.).

- Pérdida de control: los datos y la información ahora estarán alojados en otro servidor que no es nuestro.

- Pueden haber servicios que el prestador no ofrece. Esto incluye desde funcionalidades completas hasta versiones de software específicas que podrían ser necesarias para nuestro sistema.

- Dependiendo del servicio contratado, muchas de las ventajas antes mencionadas pueden estar más o menos garantizadas. Es necesario evaluar bien las prestaciones que el servicio cloud ofrece y controlar que sea lo que nuestro sistema necesita para funcionar correctamente.

9) Imagine tiene que implementar un sistema de firma digital: dado un pdf de entrada debe devolverlo firmado digitalmente. Para ello, y dado que debe integrarse a sistemas web existentes, debe diseñar una arquitectura que facilite dicha integración. Comente sobre los componentes de la misma y qué cuestiones contempla, dificultades, etc.

Para la resolución de este ejercicio voy a hacer el seguimiento de una transacción iniciada por un cliente, solicitando firma digital a nuestro sistema. Para no extenderme en exceso, voy a omitir algunos pasos de seguridad obvios para estas alturas y que considero no es el objetivo a evaluar en este punto, tales como sanear las entradas de datos, etc.:

Para efectuar la firma digital, el cliente deberá en primera instancia enviarnos el pdf. Como no sé qué tan sensible son los datos que contiene ese archivo, voy a suponer que es obligatorio que el cliente lo encripte con nuestra clave pública y con su clave privada, esto último principalmente por si es necesario garantizar en no repudio por parte del cliente. Luego, se conectará a nuestra API y nos enviará ese documento encriptado junto con el hash generado del mismo.

Una vez que estos datos estén en nuestro sistema, lo primero que haremos será desencriptar el documento con nuestra clave privada y con la clave pública del cliente; y posteriormente calcular su hash. Tras verificar que ambos hash coincidan, podemos asegurarnos que ese documento fue enviado por el cliente en cuestión. Luego, el servidor procesará el pdf verificando que todo esté en orden para poder firmarlo.

Una vez verificado esto, se encriptará el documento utilizando nuestra clave privada y la clave pública del cliente, generando también el hash correspondiente. Acto siguiente se le enviará al cliente el documento ya firmado junto con el hash y, cuando reciba estos datos, desencriptará el documento con su clave privada y con nuestra clave pública. Tras calcular el hash y compararlo con el que le hemos enviado, el documento, pues, estará firmado por nuestra entidad. El cliente puede estar seguro de que nuestro sistema ha comprobado y verificado que el documento es auténtico y que lo hemos firmado en consecuencia. Todo esto gracias a que, como fue encriptado por nuestra clave privada, sólo nosotros podemos generar ese documento.

Ese sería el proceso básico y el mejor caso posible. Pero podrían presentarse algunos problemas:

- Si el hash recibido y el calculado no coincide (independientemente del origen-destino), todo el contenido se descarta y se solicita una retransmisión.

- Si nuestro sistema, al procesar el pdf detecta algún tipo de irregularidad (que dependerá del servicio que debe brindar), el documento se descarta (pero no se borra) y se notificará al usuario, logeando todo el proceso.

- Como el enunciado no especifica lo que debe hacer para comprobar que un documento es válido y firmarlo, no puedo dar más detalles sobre lo que debe hacer para discernir entre un documento válido y uno inválido.

- Que el usuario sea o no válido dependerá del sistema de login implementado.

10) Suponga que está desarrollando una API que puede ser consumida utilizando diferentes formatos de intercambio de datos ¿De qué forma puede

determinar el backend el formato a utilizar para atender un cliente determinado? ¿Cómo debería comportarse el mismo en caso de no conocer el formato solicitado?

Cuando el cliente hace una petición a nuestra API, puede establecer un valor en el header de la petición HTTP, específicamente en el campo “accept”. Este campo determina los formatos MIME que el cliente acepta. Será la forma de indicarle al servidor el formato en el que se desea la respuesta.

Cuando el cliente envíe el HTTP request al servidor, este último deberá recuperar el header de la petición y, mediante un “Switch-case”, evaluar qué acción tomar en base a su contenido. Deberá haber un “case” por cada formato MIME que el servidor acepte. En caso de que el formato solicitado coincida con alguno de los formatos que el servidor ofrece, se le entregará al cliente el código 200 (ok) y el recurso correspondiente en dicho formato. Pero en caso de que el formato no coincida con ninguno de los aceptados, el servidor retornará un error 406 (Not Acceptable), el cual indica al cliente que el servidor no es capaz de devolver los datos en ninguno de los formatos solicitados.