1. **Miembros del equipo**
   - Catalina Patiño Forero (Disponibilidad)
   - Juan Pablo Calad Henao (Rendimiento)
   - Mateo Agudelo Toro (Seguridad)

2. **Diseño de arquitectura de la Aplicación y Sistema**
   a. **Vista de desarrollo**
      i. **Definición de Tecnología de Desarrollo**
      Para el desarrollo de la aplicación como tal se usa Angular para el frontend, NodeJS para el backend, y MongoDB para el almacenamiento de datos. La autenticación es completamente tercerizada utilizando los servicios proveídos por Auth0 (que a su vez permite la integración con múltiples servicios de autenticación como Google+, Facebook o Microsoft Active Directory, Multi-Factor authentication, entre otros). También se usa Docker para crear una imagen de la aplicación (la cual se puede encontrar en el Docker Hub). El despliegue se hace sobre Kubernetes, utilizando nuestra imagen de Docker para la aplicación y otra de Nginx para el proxy inverso.
      ii. **URL del repositorio:** https://github.com/agudelotmateo/tracker-v2-auth0
   b. **Vista de despliegue**
      i. **Definición de Tecnología – Infraestructura TI: Servidores, Software base, Redes, etc.**
      La base de datos se encuentra en modo de replicación en las tres máquinas del DCA que fueron asignadas a cada uno de los miembros del equipo y la aplicación está desplegada en un cluster de Kubernetes de la universidad en el cual se nos fue asignado un namespace.
      ii. **URL de ejecución pública y segura:** http://proyecto2.dis.eafit.edu.co/

3. **Implementación y Pruebas por Atributo de Calidad**
   a. **Disponibilidad**
      i. **Implementación**
         1. **Herramientas utilizadas**
         La herramienta principal es Kubernetes, donde los manifiestos se crearon en base al ejemplo dado para la clase (https://github.com/st0263eafit/appwebArticulosNodejs/). Para crear la imagen usada por Kubernetes se usa docker, especificamente, se crea un Dockerfile que define una imagen con nuestra aplicación sobre node. Además en los manifiestos de kubernetes se especifica el uso de una imagen para nginx, que presta el servicio de proxy inverso. Por último, para la persistencia de los datos la aplicación se conecta a mongo, encontrado en tres máquinas virtuales, las cuales fueron configuradas para crear un replica set (con un nodo primario y dos secundarios), de forma que se da replicación de la información a todas las máquinas, y en caso de que falle el nodo primario, uno de los secundarios toma su lugar.

Por otro lado, se crea un docker-compose usado para desarrollo, en este se manejan tres contenedores: node + aplicación, nginx y mongo.

2. **Cambios en la implementación de la aplicación**
   En la implementación de la aplicación como tal, solo se modifica la URI usada para la conexión con mongo en el backend, de forma que se especifican las IPs de las tres máquinas que hacen parte del replica set.

**ii. Esquemas de pruebas para comprobar el Atributo de Calidad**

1. Para verificar el funcionamiento correcto del replica set se usa robomongo. Se verifica que la creación de un registro en una base de datos se replica a las demás. Y se comprueba que al "matar" el servicio de mongo (service mongod stop) en el nodo primario, un nuevo nodo retoma la función y se hace la replicación al nodo caído cuando regresa.
2. Con docker se verifica correcto funcionamiento de la imagen definida por el Dockerfile tanto en computador local como en una de las máquinas virtuales del DCA.
3. Al verificar correcto funcionamiento de la imagen, ya en kubernetes solo queda verificar que la aplicación se ejecute en el dominio dado (proyecto2.dis.edu.co).

**b. Rendimiento**

**i. Implementación**

1. **Herramientas utilizadas** la herramienta que se usó para medir el rendimiento de la aplicación fue apache jmeter, una aplicación que usualmente es utilizada para medir y analizar el rendimiento de varios servicios, el cual en este caso se usó para hacer pruebas unitarias sobre consulta sobre la base de datos e inserciones, además se usó para medir la carga de las páginas como si muchos usuarios estuvieran usando la app al tiempo.
2. **Cambios en la implementación de la aplicación**
   El cambio más importante que se realizó en la aplicación fue la mejora en un método el cual se encargaba de guardar la localización en la cual el usuario está cada segundo, esta cambiara o no, lo que se hizo fue que se puso una tolerancia de cambio, con esto solo se guarda la localización cuando se obtiene un cambio mayor a esta tolerancia, esto nos mejora el rendimiento a la hora de guardar y consultar en la base de datos, dado a que esto evita en gran parte la redundancia de datos.

**ii. Esquemas de pruebas para comprobar el Atributo de Calidad**

- las pruebas para este atributo se hicieron usando jmeter, realizando múltiples inserciones de datos y consultas a la base de datos, esto para ver qué tanto de esto concurrentemente soporta la aplicación en un tiempo dado.
- dado a que para realizar este tipo de pruebas es algo complicado, dado a que se hicieron pruebas en jmeter no dinámicas, se hizo el

supuesto que cierto porcentaje de estos datos (localizaciones) no variarán en un tiempo dado, con la modificación realizada en el código, esto no guardaría datos redundantes

c. **Seguridad**
   i. **Implementación**
      1. **Herramientas utilizadas:** lo que se hizo fue integrar Auth0 a la aplicación para la parte del login, de forma que es completamente tercerizado y se administra desde un panel de control proveído por Auth0 y sigue funcionando de forma transparente para la aplicación.
      2. **Cambios en la implementación de la aplicación:** la aplicación fue reescrita completamente con el fin de integrarla correctamente con Auth0. Esto se hizo debido a que integrar una aplicación hecha con el framework MEAN con Auth0 es mucho más difícil que integrar una aplicación hecha con plantillas HTML. Esto se debe a que el front-end hecho en Angular se identifica como una aplicación completamente independiente al back-end, lo que generaría problemas para la generación y transmisión del token de autenticación entre ambas partes debido al CORS. Si bien este problema parecería simple de solucionar configurando el CORS (que de hecho no es tan fácil como suena) se estaría poniendo en riesgo la seguridad de la aplicación. Otra solución sería configurar un proxy con este fin, pero la solución recomendada desde prácticamente todos los puntos de vista (eficiencia, seguridad, buenas prácticas, etc) es integrarlo desde el front-end. Para lograr esto eran necesarios muchos cambios en la aplicación, por lo que me pareció una mejor idea reescribirla desde cero mejorando también otros aspectos del código (especialmente el orden y la integración con el back-end).
   ii. **Esquemas de pruebas para comprobar el Atributo de Calidad**
      1. **Verificación del esquema de autorización:** mediante el uso Postman se verificó que cada una de las rutas del API estuvieran protegidas, es decir, que no fuera posible acceder a ellas sin pasar el token de autenticación en el header correspondiente.
      2. **Verificación de la protección contra ataques XSRF:** al almacenar los tokens de autenticación en el local storage en vez de mediante el uso de cookies, no se es vulnerable a este tipo de ataques. Se puede encontrar más información con respecto al tema aquí: https://auth0.com/docs/security/store-tokens#cookie-disadvantages
      3. **Verificación de la protección contra ataques XSS:** debido al uso de Angular para la implementación del front-end de nuestra aplicación, no fue necesaria la inclusión de ningún tipo de protección contra este tipo de ataques (una de las vulnerabilidades del uso de autenticación basada en tokens). Esto se debe a que Angular se encarga de este aspecto; más información puede ser encontrada en el siguiente enlace: https://angular.io/guide/security. El único paso adicional que se necesita para estar debidamente protegido ante este tipo de ataques es el uso de HTTPS, lo cual no fue posible debido a la falta

de permisos que teníamos en el cluster de kubernetes. En más detalle, se tuvieron los siguientes problemas:

a. **kube-lego:** esta es la opción tradicional para la administración de certificados TLS en clusters de kubernetes, pero cuyo desarrollo de ha estancado y tiene como predecesor a *cert-manager*. Aun así, se intentó usar kube-lego, pero debido a la incapacidad de crear un config map en el namespace kube-system debido a la falta de permisos, fue imposible.
b. **cert-manager:** esta opción requiere permisos para crear jobs y secrets, los cuales tengo.
c. **Tradicional:** se intentó generar los certificados solo una vez (nada de que se renovaran automáticamente ni cosas por el estilo), tanto con herramientas tradicionales (no orientadas a kubernetes, como certbot) como creando self-signed certificates, pero fue imposible debido a la falta de permisos para correr comandos dentro de los pods.

**4. Marco-referencia-v3: Apropiación de las bases conceptuales, patrones, mejores prácticas, etc de los atributos de calidad seleccionados.**

## Disponibilidad

Alta disponibilidad es la habilidad de un sistema de estar disponible continuamente a los usuarios sin ninguna pérdida en el servicio. La disponibilidad indica el tiempo total que la aplicación o servicio está disponible para cumplir con los requisitos del usuario. En general, se busca alta disponibilidad dado que ésta implica mayores ingresos y ventaja competitiva para el negocio. Por otro lado, la alta disponibilidad puede ser más un requisito, dependiendo de la naturaleza del negocio como en el caso de la banca.

Los siguientes son puntos claves para tener una arquitectura de alta disponibilidad:

- Se deben satisfacer los acuerdos de nivel de servicio (SLAs).
- Debe tener estrategias de *failover* y *failback,* las cuales son transparentes para el usuario final y no representan ninguna pérdida en la información.
- Debe incluir una estrategia de monitoreo proactivo para facilitar la autodetección y autocorrección.
- Debe incluir estrategias para el manejo de *downtime* (tiempo en el que el servicio no se encuentra disponible), ya sean planeados o no planeados.
- Debe comprender estrategias para el manejo de todos los posibles puntos de fallo.

- Debe ser una infraestructura robusta con suficiente redundancia para satisfacer la disponibilidad definida en los SLAs.

La alta disponibilidad es planeada, para ello se debe analizar cómo afecta el *downtime* al negocio. Luego se especifica el desempeño SLA para disponibilidad, es decir, cuánto es el tiempo máximo permitido para una caída del sistema en el año. Por ejemplo, un SLA del 99.999% indica que el sistema sólo puede tener un *downtime* de 5 minutos al año. Para este cálculo se introducen métricas como *Recovery Point Objective* (RPO) que indica el volumen de datos en riesgo de pérdida que la organización considera tolerable, y el *Recovery Time Objective* (RTO) que expresa el tiempo máximo esperado que debe demorar un proceso en ser restablecido.

Se debe tener en cuenta que la disponibilidad de un sistema o de una aplicación suele depender de muchos sistemas y redes intermediarias que participan en la caden de entrega. Al entender estos intermediarios es posible crear un plan de optimización de los mismos, para aumentar la disponibilidad de la aplicación.

- Sistemas internos: Sistemas que hospedan las aplicaciones de la empresa, y cuyo mantenimiento dependen de la misma empresa. Son los principales responsables de la disponibilidad y desempeño de la aplicación. Ejemplo: servidores web, infraestructura de la red, bases de datos de la empresa, entre otros.
- Interfaces internas: Sistemas de integración de componentes como servicios internos. Su función principal es proveer de servicios y otras funcionalidades o datos del negocio críticos a las aplicaciones empresariales. Ejemplos: ERT, sistemas de reportes, sistemas de colaboración y sistemas de análisis.
- Interfaces externas: Sistemas que integran componentes externos con internos, por ejemplo cuando se necesita de redes sociales externas, o servicios web externos. Esto se logra a través de APIs, aplicaciones del lado del cliente, plug-ins, y servicios.
- Nube externa: Cuando una aplicación o algunos de sus componentes se hospedan en la nube. La nube provee un alto rango de disponibilidad, escalabilidad y rendimiento.
- Red de entrega de contenidos (CDN): Facilitan almacenamiento en caché del contenido de forma que se puede dar una entrega óptima al usuario final.
- Sistemas del usuario final: Dispositivos y software que usa el usuario final para acceder a la aplicación.

Para determinar la disponibilidad de un sistema, se deben seguir los siguientes pasos:

1. Definir la disponibilidad de los sistemas actuales.
2. Determinar cuales son las fortalezas, debilidades, oportunidades y amenazas de los sistemas actuales. Esto incluye determinar la vulnerabilidades y las fallas en seguridad de las aplicaciones.
3. Optimizar los problemas que aparecen en el paso anterior. Para ello se usan recomendaciones y mejores prácticas para mejorar la disponibilidad.
4. Definir la gobernanza de la disponibilidad.

Algunos de las métricas principales para definir SLA son:

- Disponibilidad del sistema: Tiempo total de actividad por periodo de tiempo dado.
- Tiempo promedio de recuperación (MTTR): Tiempo promedio que demora la recuperación de un sistema cuando se presenta una falla.
- Tiempo promedio entre fallas (MTBF).
- Disponibilidad del servicio: Tiempo total en el que los servicios del negocio están disponibles para un periodo de tiempo dado.

Cuando se busca alta disponibilidad es muy probable encontrar los siguientes retos:

- Retos relacionados con el hardware: Infraestructuras no escalables, retos con el ancho de banda, fallas en el hardware, existencias de puntos únicos de fallas (componentes de los que depende todo el sistema), o existencia de cuellos de botellas no escalables.
- Retos relacionados con el software: Mal diseño, código mal escrito, fallos en la seguridad, falta de estrategias de uso de caché, ausencia de pruebas de disponibilidad.
- Retos relacionados con los procesos: Ausencia de gobernanza de la disponibilidad de los procesos, ausencia de monitoreo y notificación de los procesos.
- Retos no anticipados: Desastres naturales, incidentes de seguridad.

**Patrones de arquitectura**

- *Failover*: Es la habilidad de un sistema de mantenerse operacional en el evento de una falla de un nodo o un componente gracias al uso de un componente de respaldo. Se suele implementar usando un cluster de nodos, donde cada uno se componen de código y configuración similar.

- *Failback*: Suele pasar después de un *failover* y es la recuperación de un sistema a un estado totalmente operacional cuando ocurre una falla. Esto requiere de establecer un punto de falla y copiar los datos que fueron creados después de la falla del nodo de respaldo.
- Replicación: Involucra copiar los datos del nodo primario a todos los nodos de respaldo para facilitar el cambio en caso de *failover*. Hay dos tipos de replicación: activa, donde la solicitud del cliente es procesada por todos los nodos (usada en sistemas de tiempo real), y pasiva, el nodo primario procesa la solicitud y luego se copia a los nodos secundarios.
- Redundancia: Se logra al introducir nodos de espera (*standby*) que se encargan de los procesos cuando ocurre un *failover*.
- Virtualización: Permite la distribución de carga y ruteo de solicitudes, además permite a los administradores del sistema mejorar el hardware sin problemas.
- Mantenimiento continuo: Éste incrementa la confiabilidad del hardware y las operaciones.

**Patrones de software**

1. Patrón de degradación de la funcionalidad elegante y paso a paso: Evitar la pérdida de la funcionalidad por completo, sino que degradarla de forma “elegante”, para ejecutar procesos más cortos.
2. Asincronía y servicios basados en integración con interfaces externas: Se reduce el acoplamiento con las interfaces externas usando un patrón de integración asincrónica (AJAX). Esto permite cargas de páginas no bloqueantes y previene la pérdida de otras funcionalidades no relacionadas en la misma página.
3. Componentes de aplicación sin estado y ligeros: Entre menos estados tenga y más ligera sea la aplicación, se puede replicar más veces. Además pueden ser fácilmente compartidas y sincronizadas, haciendo de la aplicación más robusta y confiable.
4. Código de incremento continuo y replicación de los datos: Permite consistencia y reduce la pérdida de información.
5. *Trade-off* de disponibilidad usando el teorema CAP: El teorema establece que no es posible lograr disponibilidad (asegurar que los clusters están siempre operacionales), consistencia (consistencia de los datos en todos los cluster) y tolerancia a las particiones (el cluster es operacional sin importar las particiones en las redes) simultáneamente. Por tanto, de acuerdo a los requisitos de la aplicación, se puede hacer un *trade-off* entre los parámetros.

    - Disponibilidad y tolerancia a particiones son principales: se usan bases de datos no relacionales. Ejemplo: blog, wiki, posts y escenarios de BigData.
    - Disponibilidad y consistencia son principales: se usan sistemas centralizados como RDBMS. Ejemplo: e-commerce y aplicaciones financieras.
    - Tolerancia a particiones y consistencia son principales: Se logra usando MongoDB, Redis, BigTable, MemcacheDB. Aparece cuando los datos deben ser consistente a lo largo de todos los nodos en el cluster, y el cluster debe tener tolerancia a las particiones.

**Mejores prácticas**

- Relacionadas con el hardware:
    - Crear una infraestructura proactiva interna y externa para el monitoreo y notificación.
    - Emplear redundancia del hardware.
    - Asegurar la existencia de un lugar de recuperación de desastres, el cual debe tener una réplica del código y los datos del lugar principal.
    - Minimizar el *downtime* en cortes planeados usando componentes de respaldo.
- Relacionadas con el software:
    - Mantener la arquitectura lo suficientemente simple para que sea extensible y mantenible.
    - Diseñar componentes de software modulares, para que sea fácilmente escalable.
    - Diseñar una estrategia de almacenamiento en caché comprensiva, que asegure que los datos frecuentemente utilizados se encuentren en caché.
    - Evitar conversaciones inoficiosas con servicios superiores para minimizar la transferencia de datos por solicitud.
    - Adoptar continuamente pruebas de seguridad y regresión para encontrar fallas o huecos en la seguridad tempranamente.
    - Maximizar la automatización para actividades de mantenimiento como actualizaciones de software/aplicaciones.
    - Construir rutinas de manejo de error en el código.
    - Establecer lineamientos bien definidos para la clasificación de los datos para prevenir cualquier accidente y exposición de datos sensibles.

    - Probar todos los posibles escenarios de disponibilidad y verificar los SLAs.
- Relacionados con los procesos:
    - Formular un proceso comprensivo de gobernanza de la disponibilidad.

**Modelo de las 5R**

- Confiabilidad (*Reliability*): Es la probabilidad de que un sistema no tenga fallas. La confiabilidad del sistema depende de la confiabilidad del software, que consiste en la predicción, prevención, detección y tolerancia de fallos, y el hardware, alcanzada al proveer conexiones SAN redundantes, clusters de espera, replicación de discos, conexiones a la red redundantes, fuentes de poder redundantes
- Replicabilidad (*Replicability*).
- Recuperación (*Recoverability*): Indica que tan bien se recupera el sistema de un escenario de error. Se tiene en cuenta la tolerancia a fallos, alcanzada con degradación de la funcionalidad, y balanceamiento transparente de la carga, logrado con algoritmos de balanceo. También se debe tener en cuenta que tan bien el sistema provee un *failover* transparente y qué tanto tiempo necesita el sistema para recuperarse.
- Reporte y monitoreo (*Reporting and monitoring*).
- Redundancia (*Redundancy*): Se alcanza con una buena planificación y dimensionamiento de la infraestructura/capacidad y acomodando módulos y sistemas de espera para recibir los datos en caso de falla. Se debe aplicar en diferentes niveles de infraestructura: componentes, redes, aplicación, sistema, entre otros.

**Bibliografía**

http://proquestcombo.safaribooksonline.com.ezproxy.eafit.edu.co/book/software-engineering-and-development/enterprise/9780128022580/firstchapter

## Rendimiento

Web performance optimization **(WPO)** es esencialmente métodos y técnicas para optimizar la velocidad de páginas web, que implica analizar los componentes de las

mismas para optimizar su tiempo de respuesta, esto envuelve también otros factores que impactan la velocidad directamente o indirectamente, tales como los componentes de infraestructura, monitoreo y mantenimiento de estos componentes y estrategia de gestión de contenido. En general el rendimiento es la habilidad de tiempo de respuesta cuando ocurre un evento ya sean interrupciones, mensajes, una petición de usuario o algún otro evento del sistema, donde se debe responder a él en cierto tiempo, el tiempo en el que este evento responde es la esencia de rendimiento. Algo de lo más importante del rendimiento es el control de la combustión interna, para esto el sistema debe maximizar la potencia y eficiencia y minimizar la combustión, por esto el sistema mide su rendimiento en número de transacciones por minuto.

**Concurrencia**

Un escenario de rendimiento comienza con un evento que llega al sistema. Respondiendo correctamente al evento, requiere que se consuman recursos. Mientras esto sucede, el sistema puede estar atendiendo simultáneamente otros eventos.
Un serie de eventos pueden llegar en patrones predecibles o distribuciones matemáticas, o ser impredecibles. Estas llegadas son caracterizadas como periódicas, estocásticas o esporádicas.
Dicho anteriormente el modo de llegada de los eventos al sistema, este puede responder de diferentes maneras y la respuesta del sistema a estas llegadas, son medidas de las siguientes maneras:

- Estado de latencia: El tiempo entre la llegada del estímulo y la respuesta del sistema a él.
- El rendimiento del sistema: usualmente se da como el número de transacciones que el sistema puede procesar en una unidad de tiempo.
- La fluctuación de la respuesta: La variación permitida en la latencia.
- La cantidad de eventos no procesados porque el sistema estaba demasiado ocupado para responder.

**Análisis**

Por las anteriores consideraciones ahora describiremos individualmente cada una de los escenarios generales para rendimiento.

- **Fuente de estímulo:** Los estímulos llegan de fuentes externas (posiblemente múltiples) o internas.
- **Estímulo:** Los estímulos son los eventos que llegan. El patrón de llegada puede ser periódico, estocástico o esporádico, caracterizado por parámetros numéricos.
- **Artefacto:** El artefacto es el sistema o uno o más de sus componentes.
- **Ambiente:** El sistema puede estar en varios modos de operación, como normal, emergencia, carga pico o sobrecarga.
- **Respuesta:** El sistema debe procesar los eventos que llegan. Esto puede causar un cambio en el entorno del sistema.
- **Medida de respuesta:** Las medidas de respuesta son el tiempo que toma procesar los eventos que llegan (latencia o una fecha límite), la variación en este momento, la cantidad de eventos que pueden procesarse dentro de un intervalo de tiempo particular (rendimiento) o una caracterización de los eventos que no pueden ser procesados (tasa de error).

**Tácticas para un mejor rendimiento**

El objetivo de las tácticas de rendimiento es generar una respuesta a un evento que llegue al sistema dentro de alguna restricción basada en el tiempo.
El evento puede ser único o una secuencia. Las tácticas de rendimiento controlan el tiempo dentro del cual se genera una respuesta a estos.
En algún momento durante el periodo después que el evento llegue pero antes de que el sistema lo responda, el sistema está trabajando para responder a ese evento o el procesamiento está bloqueado por algún motivo. Esto lleva a los dos contribuyentes básicos al tiempo de respuesta: el tiempo de procesamiento (cuando el sistema está trabajando para responder) y el tiempo bloqueado (cuando el sistema no puede responder).

- Tiempo de procesamiento: El procesamiento consume recursos, lo que lleva tiempo. Los eventos se manejan mediante la ejecución de uno o más componentes, cuyo tiempo empleado es un recurso. Los recursos de hardware incluyen CPU, almacenamiento de datos, ancho de banda de comunicación de red y memoria. Los recursos de software incluyen entidades definidas por el sistema en diseño.

- Tiempo de bloqueo: Un cómputo se puede bloquear debido a la contención de algún recurso necesario, porque el recurso no está disponible o porque el cálculo depende del resultado de otros cálculos que aún no están disponibles.
- Controlar la demanda de recursos: Esta táctica opera por el lado de la demanda para producir una menor demanda de los recursos que tendrán que servir los eventos.
- Administrar recursos: Esta táctica opera en el lado de la respuesta para hacer que los recursos a mano funcionen de manera más efectiva en el manejo de las demandas planteadas.

**Claves para un alto rendimiento:**

- Asignación de responsabilidades: determina las responsabilidades del sistema que implica una carga pesada, tener requisitos de respuesta críticos en el tiempo, son muy usados, o partes de impacto del sistema donde ocurren cargas pesadas o eventos críticos en el tiempo. Para esto se sugiere lo siguiente:
    - Responsabilidades que resultan de un hilo de control a través de un proceso o límites del procesador.
    - Responsabilidades en el manejo de hilos de control, asignación o desasignación de hilos, mantenimiento de hilos.
    - Responsabilidades para programar recursos compartidos o administrar artefactos relacionados con el rendimiento, como las colas, buffers y caches.

- Modelos de coordinación: determina los elementos del sistema que deben coordinarse con otros directamente o indirectamente y escogen mecanismos de comunicación y coordinación que hacen lo siguiente:
    - Soporte a cualquier concurrencia introducida.
    - Asegurar que la respuesta de rendimiento requerida pueda ser entregada.
    - Puede capturar eventos periódicos, estocásticos o esporádicos que lleguen como necesidad.

- Modelos de datos: determine aquellas partes del modelo de datos que implica una carga pesada, que tengan requisitos de respuesta críticos en el tiempo, que se utilicen ampliamente o que afecten a partes del sistema

donde ocurren cargas pesadas o eventos críticos. Por esto se determina lo siguiente:
    - Si se mantienen copias múltiples de datos clave se beneficia el rendimiento.
    - Si se hace un particionamiento de los datos se beneficia el rendimiento.
    - Si se reduce el procesamiento para la creación, inicialización, manipulación, traslación y destrucción de innumerables datos.

- Gestión de recursos: determinar cuáles recursos en el sistemas son críticos para el rendimiento. por estos recursos, se asegura que estos sean monitoreados y gestionados bajo operación normal y sobrecargada del sistema. por ejemplo:
    - Procesos / Modelos de hilos.
    - Priorización de recursos y acceso a recursos.
    - Estrategias de programación y bloqueo.
    - Despliegue adicional de recursos.

- Tiempo de enlace: para cada elemento que se vincula después del tiempo de compilación se determina lo siguiente:
    - Tiempo necesario para completar el enlace.
    - Gastos indirectos adicionales introducidos mediante el uso del mecanismo de enlace tardío.

- Elección de tecnología: elecciones de tecnología que permitirán establecer y cumplir plazos duros y en tiempo real:
    - política de programación
    - prioridades
    - políticas para reducir la demanda
    - asignación de partes de la tecnología a los procesadores
    - otros parámetros relacionados con el rendimiento

**Bibliografía**

http://ezproxy.eafit.edu.co:2288/book/software-engineering-and-development/9780132942799/firstchapter

## Seguridad

La seguridad mide la habilidad del sistema para proteger datos y servicios del acceso no autorizado sin dejar de proveer acceso a las personas y otros sistemas que sí están autorizadas. Debe proteger tanto los datos almacenados como los que son transferidos y los demás procesos que actúan sobre estos.

Un ataque se define como una acción contra un sistema (computacional) con el fin de reducir uno de los siguientes aspectos de este:

- **Confidencialidad:** los datos y servicios están protegidos contra el acceso no autorizado.
- **Integridad:** los datos y servicios no sufren de manipulación por entes no autorizados.
- **Disponibilidad:** el sistema estará disponible para uso justo (sin mala intención).

Una estrategia de seguridad exhaustiva debe hacerse cargo de estos tres elementos en todas las capas, para todos los componentes y contra cualquier tipo de amenaza de seguridad. Estas categorías se apoyan en:

- **Autenticación:** verifica la identidad de las partes involucradas en una transacción.
- **No repudio:** garantiza que el remitente de un mensaje no puede posteriormente negar el envío del mensaje, al mismo tiempo que quien recibe no puede negar haber recibido el mensaje.
- **Autorización:** otorga al usuario legítimo el privilegio de realizar una tarea.

Definición de otros términos:

- **Vulnerabilidad:** falla o debilidad del sistema o de la aplicación que permite que sucedan incidentes de seguridad.
- **Amenaza:** agente responsable del incidente de seguridad.
- **Riesgo:** probabilidad de que ocurra un ataque multiplicado por el potencial impacto de este.

Para mejorar la seguridad de un sistema se deben seguir los siguientes pasos:

1. **Análisis de seguridad:** se consideran los riesgos y se definen las políticas para enfrentarlos.
2. **Modelado de amenazas:** se modelan las posibles amenazas con el fin de analizarlas mejor y se clasifican con base a la probabilidad de que ocurran.
3. **Diseño:** basándose en los elementos obtenidos en los dos pasos anteriores se mapea una política de seguridad para cada posible escenario, la cual

debe estar orientada a la prevención, detección y recuperación ante cada incidente de seguridad.

4. **Implementación:** teniendo listo el diseño este debe implementarse cuidadosamente en todas las capas y para todos los componentes tanto de software como de hardware que conforman la aplicación.
5. **Pruebas:** se deben identificar los nuevos problemas potenciales de seguridad ya sea mediante las diferentes herramientas disponibles con esta finalidad y/o expertos en seguridad.
6. **Monitoreo:** la seguridad de la aplicación debe considerarse a cada momento, incluso luego del deployment. Las políticas deben actualizarse de forma oportuna para una máxima protección.

**Análisis (1) y modelado (2)**

En seguridad generalmente se modelan las amenazas. De un escenario general de seguridad es posible describir porciones individuales al caracterizar las siguientes categorías:

- **Fuente del estímulo:** fuente del ataque, que puede ser humana u otro sistema, desconocida o previamente identificada (correcta o incorrectamente). En caso de ser humana puede ser de fuera o dentro de la organización.
- **Estímulo:** el ataque propiamente. Son intentos no autorizados de
    - Mostrar, cambiar o eliminar datos
    - Acceder a los servicios del sistema
    - Cambiar el funcionamiento del sistema
    - Reducir la disponibilidad del sistema
- **Artefacto:** el objetivo del ataque, que pueden ser los servicios del sistema, los datos que este almacena, los datos que produce o los datos que consume.
- **Ambiente:** el ataque puede darse en diferentes situaciones:
    - El sistema está o no en línea
    - El sistema está o no conectado a una red
    - El sistema está o no detrás de un firewall
    - El sistema está completamente operacional, parcialmente operacional o completamente caído
- **Respuesta:** el sistema debe garantizar que las transacciones se hacen de forma que los datos y servicios que estas acarrean están protegidas contra accesos no autorizados.

- **Medida de respuesta:** incluye información como qué tanto el sistema está comprometido cuando un componente particular o dato es comprometido, cuánto tiempo tardó el ataque en ser detectado, cuántos ataques fueron resistidos, cuánto tiempo tomó recuperarse de un ataque exitoso y qué cantidad de datos fueron vulnerados por un ataque específico.

**Diseño (3)**

Adicional a las políticas de control se requieren los principios de seguridad, los cuales proveen los lineamientos de arquitectura que deben ser respetados tanto en el diseño de la aplicación como de la infraestructura. Estos principios se diseñan con base en las mejores prácticas, tendencias de la industria, estándares empresariales y cualquier ley y/o regulación que aplique.

- **Seguridad con defensa en profundidad**
  - **Detalles:** este principio hace cumplir las políticas de seguridad apropiadas en cada capa, componente, sistema y servicio mediante el uso de técnicas, políticas y operaciones adecuadas. Las políticas de seguridad y controles en cada capa deben ser diferentes a las de todas las demás, haciendo más difícil para un hacker comprometer el sistema completo.
  - **Justificación:** previene al potencial hacker de obtener acceso al sistema entero, pues el hecho de que una capa esté comprometida no implica que las demás también lo están.
  - **Ejemplo:** firewall, más seguridad en el servidor web, más seguridad a nivel de sistema operativo, más seguridad en el servidor de aplicaciones, más seguridad a nivel de red, más seguridad en la misma aplicación, más seguridad en la base de datos...
- **Parchar el enlace más débil**
  - **Detalles:** identifique y arregle el punto más vulnerable de la cadena de componentes que conforman el sistema.
  - **Justificación:** previene el abuso de recursos e intentos de hackeo, tanto accidentales como intencionales.
- **Principio de privilegios mínimos**
  - **Detalles:** por defecto cada rol debe tener la menor cantidad de privilegios que le sea posible para que pueda hacer correctamente su trabajo. Los privilegios no serán elevados de forma automática, ni directa ni indirectamente. La información solo será compartida en caso de ser estrictamente necesario.
  - **Justificación:** previene el abuso de recursos e intentos de hackeo, tanto accidentales como intencionales.
  - **Ejemplo:** permitir acceso de solo lectura a los archivos.
- **Compartimentación**
  - **Detalles:** todos los recursos y funciones de hardware y software deben ser categorizadas en diferentes clasificaciones de seguridad, y

el acceso a estas debe ser permitido únicamente a los usuario con los roles y privilegios apropiados.
    - **Justificación:** previene la exposición accidental de datos confidenciales y bloquea el acceso no autorizado a recursos.
- **Punto de entrada único**
    - **Detalles:** la aplicación debe permitir el acceso a los usuario a través de un único punto de autenticación. Se deben evitar tanto las entradas traseras como las URLs atajo.
    - **Justificación:** minimiza la posibilidad de un acceso no autorizado.
    - **Ejemplo:** todas las páginas privadas o protegidas redireccionan automáticamente a la página de login que es el punto único de entrada. No se permite el paso de credenciales de usuario por URL.
- **Gestión de la administración de la seguridad**
    - **Detalles:** el sistema debe tener una visión holística de las funciones administrativas con el fin de gestionar las funcionalidades importantes respecto a la seguridad.
    - **Justificación:** minimiza la posibilidad de un acceso no autorizado.
- **Soporte para la extensibilidad**
    - **Detalles:** el framework de seguridad debe ofrecer soporte al modelo de plug-ins basados en estándares en el cual es posible escribir extensiones personalizadas para mejorar la seguridad. Estas extensiones permiten usar estas mismas condiciones de seguridad en diferentes ambientes y cumplir con leyes locales y regulaciones.
    - **Justificación:** mejora la seguridad a través de extensiones personalizadas.
    - **Ejemplo:** normalmente los servidores de aplicación proveen extensiones de seguridad personalizados como diferentes módulos de login para permitir diferentes tipos de mecanismos de autenticación.
- **Validación de datos de usuario**
    - **Detalles:** los datos entrados por el usuario deben ser validados cuidadosamente y limpiados en varios niveles. De igual manera, los datos deben ser codificados de manera adecuada tanto al almacenarlos como al transferirlos entre diferentes capas.
    - **Justificación:** previene ataques causados por contenido malicioso alojado en los datos de usuario.
    - **Ejemplo:** los datos de usuario se validan tanto en el cliente como en el servidor.
- **Minimización de la superficie de ataque**
    - **Detalles:** minimice los puntos de entrada para los usuarios públicos y exponga a usuario no autorizados la menor cantidad de datos, servicios y funcionalidad posible. Se denomina superficie de ataque al código que puede ser ejecutado por usuarios sin requerir autorización.
    - **Justificación:** minimiza las oportunidades de ataque y reduce el éxito de los intentos de hackeo.

    - **Ejemplo:** las sesiones inactivas se cierran automáticamente.
- **Planeación para el fracaso**
    - **Detalles:** diseñe planes de contingencia para todos los posibles escenarios en los que pueda fallar la seguridad. Utilizando rutinas robustas de manejo de errores, copias de seguridad de los datos, un ambiente de recuperación ante desastres y defensa en profundidad es posible minimizar el impacto de los incidentes de seguridad.
    - **Justificación:** minimiza el impacto de los incidentes de seguridad y asegura la disponibilidad continua.

**Implementación (4)**

- **Detectando los ataques**
    - **Detección de intrusión:** comparación de patrones de tráfico de red o solicitudes de servicio dentro de un sistema contra un conjunto de patrones conocidos (llamados firmas) de comportamiento malicioso, los cuales están almacenados en una base de datos.
    - **Detección de denegación del servicio:** comparación de la firma de tráfico de red que entra a un sistema con los perfiles históricos de ataques conocidos de denegación de servicio.
    - **Verificación de la integridad del mensaje:** mediante el uso de checksums o valores de hash se verifica la integridad de archivos.
    - **Detección de retraso en los mensajes:** se deben analizar el tiempo que toma entregar un mensaje, ya se si, por ejemplo, este valor es muy variable, es posible que estemos ante un ataque por intermediario (man-in-the-middle attack).
- **Resistiendo a los ataques**
    - **Identificación de actores:** identificar la fuente de cualquier entrada externa al sistema.
    - **Autenticación de actores:** asegurarse de que sean quien dicen ser.
    - **Autorización de actores:** asegurarse de que los actores autenticados tengan derecho a acceder y modificar tanto datos como servicios, lo cual generalmente se logra controlando el acceso a diferentes partes del sistema.
    - **Límite de acceso:** exponer la menor cantidad posible de datos y servicios al público en general.
    - **Cifrado de datos:** los datos deben ser protegidos contra el acceso no autorizado.
    - **Separación de entidades:** se deben separar las diferentes entidades dentro del sistema ya sea físicamente o utilizando información conocida de antemano.
    - **Cambio de las configuraciones por defecto:** previene que los atacantes puedan obtener acceso al sistema utilizando información conocida de antemano.
- **Reaccionando a los ataques**

    - **Revocación del acceso:** ante un ataque o la sospecha de uno se debe limitar el acceso a recursos sensible, incluso a usuarios normalmente legítimos.
    - **Bloqueo del computador:** los computadores que presenten comportamiento sospechoso deben ser bloqueados. Como esto puede ser un error de un usuario legítimo, estos bloqueos deben ser temporales.
    - **Informar a los actores:** se debe informar a los actores que conforman el ataque que han sido identificados.
- **Recuperándose de los ataques exitosos:** se deben restablecer los servicios y auditar la situación con el fin de identificar tanto las causas como los atacantes, y entablar una acción judicial con estos y/o mejorar las defensas del sistema.

**Pruebas (5) y monitoreo (6)**
Se deben probar los siguientes aspectos,

- Autenticación de usuarios
- Permisos de acceso
- Protección de los datos
- Sesiones de usuario

Adicionalmente se debe realizar un análisis de vulnerabilidades del sistema completo, incluyendo la misma aplicación, el sistema operativo y la red (entre otros).

**Algunas amenazas**

- Desastres naturales
- Malware en general
- Contraseñas débiles
- Suplantación y robo de identidad
- Ataques por denegación del servicio (DoS y DDoS)
- Ataques por intermediario (man-in-the-middle attacks)
- Ataques por cruce de sitios (cross-site scripting attacks)
- Ataques por falsificación de solicitudes (cross-site request forgery)
- Ataques por envenenamiento de sesión y cookies
- Ataques por inyección de código (SQL, XML, desbordamiento del buffer)

**Bibliografía**

http://proquestcombo.safaribooksonline.com.ezproxy.eafit.edu.co/book/software-engineering-and-development/enterprise/9780128022580/firstchapter
http://ezproxy.eafit.edu.co:2288/book/software-engineering-and-development/9780132942799/firstchapter#X2ludGVybmFsX0h0bWxWaWV3P3htbGlkPTk3ODAxMzI5