# Patrón Gatekeeper

## Qué es

También se conoce como Guardian, es un patrón de diseño que se usa para controlar el acceso a un objeto a recurso. Su función es garantizar que se permita el acceso, al recurso bajo protección, solo si se cumplen ciertas condiciones o reglas.

Básicamente, es un guardia de seguridad que se encarga de decidir quién puede acceder a un recurso. En la práctica, es un objeto que actúa como intermediario entre el objeto protegido y los objetos que tratan de acceder a este. Antes de permitir el acceso, este objeto verifica que se cumplan las condiciones necesarias, como la autenticación y autorización.

Este patrón no se limita a la autenticación de usuarios, sino que también puede ser utilizado para controlar el acceso a recursos, servicios o sistemas.

Un ejemplo rápido de uso de este patrón es con ***saamfi***, que actúa como un punto centralizado de autenticación, y se encarga de proteger todos los sistemas de la universidad. De esta forma, se logra desacoplar la lógica de autenticación de la del resto de sistemas, y solo se necesita darle mantenimiento a un sistema de autenticación.

La decisión de usar este patrón, o no, depende de las necesidades y preferencias de diseño. Si se desea mantener la lógica de autenticación separada de la lógica de negocio, es recomendable usarlo. Es recomendable hacerlo así, porque mantenerlo desacoplado ayuda a tener una mayor flexibilidad y escalabilidad. Además, este patrón permite justo lo que se puede ver con ***saamfi***, que es la reutilización de la lógica de autenticación.

Sin embargo, si por alguna razón se quiere mantener todo junto, entonces no tendría sentido usar este patrón.

Un ejemplo de cuándo no usarlo sería cuando se tiene una aplicación, o API, muy sencilla, que no requiere mayor lógica. Esto es porque, en este tipo de aplicaciones, usar este patrón podría llevar más trabajo, requerir más tiempo, y terminar por considerarse sobrediseño.

## Contexto y problema

Cuando se tiene una API, esta maneja distintas tareas y acciones, que tienen que ver con la lógica del negocio.

Si un usuario malicioso logra obtener acceso a este sistema, podría ver cómo se maneja la parte de la autenticación y autorización, lo que le podría permitir acceder a recursos delicados, o información reservada.

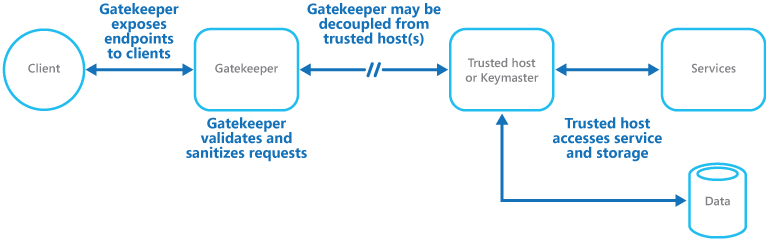
## Solución

Una solución al problema anterior sería desacoplar la lógica de autenticación de la lógica de acceso a los datos. Lo que se puede hacer es construir un objeto que funcione como una especie de fachada, por el cual se exponen los endpoints de la aplicación, y otro que se conecte y haga la comunicación con la BD y las cosas importantes.

De esta forma, el objeto autenticador no tiene acceso a las credenciales o claves para acceder al almacenamiento y a los servicios, por lo que, en caso de que corra peligro, el atacante no podría ver tales credenciales.

Este patrón actúa como un firewall en una topografía de red típica. Permite que el gatekeeper tome la decisión de dejar pasar, o no, las requests hechas al servicio. Además, permite que el gatekeeper valide y sanitice (eliminar o transformar cualquier entrada de datos que pueda ser maliciosa o insegura) el contenido de las requests antes de dejarlas pasar.

Este es un diagrama sencillo del patrón:



## Funcionamiento

La idea del patrón es que solo las solicitudes legítimas, y no maliciosas, lleguen a la capa de negocio. Su objetivo principal es mejorar la seguridad y la administración de accesos en sistemas distribuidos.

El objeto guardián (gatekeeper) actúa como una barrera de seguridad entre los clientes y los recursos protegidos. Los pasos de funcionamiento son:

* **Recepción de la solicitud:** un usuario o sistema externo trata de acceder a un recurso (API, BD, etc).
* **Autenticación:** se verifica que el usuario o sistema esté autenticado (con JWT, OAuth, API key, etc). En caso de fallar la autenticación, se rechaza la solicitud.
* **Autorización:** se comprueba que el usuario tenga permisos adecuados para acceder al recurso que solicita. Para esto, se deben definir las reglas y políticas de acceso (roles, permisos, etc).
* **Filtrado y validaciones adicionales:** se hacen validaciones adicionales (esta es la sanitización) como límite de tasa de solicitudes (rate limiting), análisis de payload, análisis para verificar que la solicitud no sea maliciosa (como un sql injection, xss, etc).
* **Encaminamiento:** en caso de que la solicitud sea válida, el guardián la pasa al servicio correspondiente.
* **Respuesta al cliente:** el servicio envía la respuesta al gatekeeper, y este se la pasa al usuario.

## Sanitización

Es un paso crítico en este patrón, pues se encarga de evitar que datos maliciosos y no autorizados lleguen a servicios internos. Se enfoca en limpiar, validar, y filtrar el contenido de las solicitudes, antes de que sean procesadas.

La idea es hacer que el sistema no sea vulnerable a ataques como:

* SQL injection: en donde los atacantes pueden hacer consultas a la BD mandando datos maliciosos.
* XSS: donde se inyectan scripts maliciosos en respuestas HTML.
* Deserialización insegura: datos manipulados pueden comprometer la ejecución del sistema.
* Path traversal: atacante puede acceder a archivos fuera del directorio permitido.

Para poder hacer esto, hace varios pasos:

1. **Validar estructura de la request**: verifica que la request tenga solo los campos esperados y que cumpla con el formato correcto. Así, se evita que se manden datos inválidos o con tipos incorrectos.

Esto es útil porque se puede provocar un error en el sistema, al guardar datos con tipos malos o así. Por ejemplo, si no se verifica esto, y se le manda una letra a un campo que debería contener solo números, puede haber un error cuando se traten de usar esos valores como si fueran numéricos.

1. **Filtrar contenido malicioso:** se eliminan patrones o caracteres que podrían usarse en ataques.

Por ejemplo, se puede definir un patrón para determinar si una cadena busca ejecutar un XSS. Al recibirla, se pasa por el patrón y, en caso de dar positivo, rechaza la request.

1. **Limitación de tamaños y estructuras:** se establecen límites de tamaño de los campos y se bloquean los payloads que sean sospechosos.

## Problemas y consideraciones

Hay que tener en cuenta algunas cosas a la hora de implementar este patrón:

* El trusted host solo debe exponer endpoints internos usados por el gatekeeper. No debe exponer nada que pueda ser accedido de otra manera que no sea pasando a través de este.
* El gatekeeper debe ejecutarse en un modo de privilegios limitado, lo que suele requerir que corra en un servicio de host separado.
* El gatekeeper solo debe validar y sanitizar las peticiones. No debe hacer ninguna acción de procesamiento distinta, ni acceder a algún tipo de data.
* El trusted host puede necesitar hacer validaciones adicionales sobre las peticiones, pero el gatekeeper debe hacer la validación básica.
* Se debe usar un canal de comunicación seguro (HTTPS, SSL, or TLS) entre el gatekeeper y el trusted host.
* Este patrón puede introducir un poco de latencia, o empeorar un poco el performance del servicio, debido al procesamiento adicional, y la comunicación entre redes requerida.
* El gatekeeper es, básicamente, un punto central de fallo, por lo que es recomendable tener una redundancia y asegurar la escalabilidad del sistema, para asegurar la disponibilidad siempre.

## Cuando usarlo

Este patrón puede servir para aplicaciones que:

* Manejan información sensible.
* Exponen servicios que requieren un alto grado de protección contra ataques maliciosos.
* Hacen operaciones críticas, e importantes, que no pueden interrumpirse.
* Requieren que las acciones de validación de las peticiones se hagan de forma separada de las tareas principales.
* Requieren, o quieren, simplificar el mantenimiento y administración del sistema de autenticación.

## Cuando no usarlo

Este patrón no es recomendable para aplicaciones que:

* Son triviales, o muy sencillos, y no requieren una gran cantidad de lógica para la autenticación y autorización.
* Necesitan una seguridad baja.
* Tienen reglas de autenticación y autorización simples (como solo login).
* No están pensados para escalar o crecer.
* Se quiere mantener todo el sistema acoplado.

## Conceptos clave

**Trusted host:** es el servicio que está detrás del gatekeeper. Se le dice así porque la aplicación corre en modo confianza, es decir, que se le dan todos los privilegios de acceso a todos los recursos.