Arquitetura de Software

Estratégias para Autenticação e Autorização Segura

Prof. Klayton R. Castro

IDP
Instituto Brasileiro de Ensino, Desenvolvimento e Pesquisa

1 de outubro de 2024

Autenticação: O que é?

- Autenticação é o processo de verificar a identidade de um usuário ou sistema.
- Pergunta fundamental: "Quem é você?"
- Fatores de autenticação:
  - Algo que você sabe (senha, PIN)
  - Algo que você tem (token, smart card)
  - Algo que você é (biometria)
- Aplicação na arquitetura de software:
  - Frontend: Coleta credenciais do usuário.
  - Backend: Valida credenciais e gera tokens de autenticação.

Autorização: O que é?

- Autorização é o processo de verificar o que o usuário pode fazer após a autenticação.
- Pergunta fundamental: "O que você pode fazer?"
- Tipos de autorização:
  - Controle baseado em regras (RBAC): Permissões baseadas em papéis (admin, usuário, etc.)
  - Controle baseado em atributos (ABAC): Ações baseadas em atributos do usuário.
- Aplicação na arquitetura de software:
  - Verificar permissões de acesso a recursos ou rotas no backend.
  - A comunicação entre diferentes camadas do sistema pode incluir tokens JWT para autorização.

Autenticação vs. Autorização

- **Autenticação** verifica quem você é, enquanto **Autorização** verifica o que você pode fazer.
- Ambas devem ser tratadas de forma independente na arquitetura do sistema.
- Exemplo prático:
  - **Autenticação**: Um usuário faz login com nome de usuário e senha.
  - **Autorização**: Após login, o sistema verifica se o usuário tem permissão para acessar um recurso específico.
- Arquitetura modular facilita a manutenção e expansão dos sistemas que implementam ambos os processos.

Princípios de Implementacao

- Ate agora vimos que uma boa arquitetura preza por:
  - **Modularidade**: Dividir o sistema em componentes independentes que possam ser desenvolvidos, testados e mantidos separadamente.
  - **Separação de Preocupações**: Manter a lógica de autenticação e autorização separada da lógica de negócios.
  - **Flexibilidade e Escalabilidade**: Facilitar a adição de novos serviços, como múltiplos métodos de autenticação.
- Exemplo: Um sistema pode ter componentes para gerenciamento de usuários, autenticação, autorização e recursos, cada um separado, mas interligado.

Camadas de Aplicação na Arquitetura

- **Frontend**: Responsável por coletar credenciais e dados do usuário.
- **Backend**: Valida autenticação e gerencia a autorização de recursos.
- **Banco de Dados**: Armazena dados de usuários e suas permissões (credenciais hash, roles, etc.)
- **Camadas de Serviço**: Implementam regras de negócio e verificam permissões.
- **Comunicação entre camadas**: JWT pode ser usado para compartilhar informações de autenticação entre as camadas.

Factory Pattern: Criação de Componentes de Autenticação

- O Factory Pattern permite criar instâncias de objetos de autenticação de forma flexível.
- Vantagens:
  - Facilita a troca de estratégias de autenticação (senha, token, biometria) sem alterar a lógica do sistema.
  - Promove desacoplamento entre a lógica de autenticação e o backend.
- Exemplo: Um factory que gera objetos de autenticação dependendo do tipo (JWT, OAuth, etc.).

Decorator Pattern: Proteção de Rotas

- O Decorator Pattern é útil para adicionar responsabilidades a funções, como verificar a autenticidade do token JWT antes de permitir o acesso a uma rota.
- Vantagens:
  - Facilita a proteção de rotas sem modificar o código principal das funções.
  - Reuso do código de autenticação em diversas partes do sistema.
- Exemplo: Um decorator em Flask que verifica a validade do token JWT antes de permitir a execução de uma função.

Comparação: Sessões vs. Tokens

- **Sessões**:
  - Mantidas no servidor, identificando o usuário autenticado com um ID de sessão.
  - Utiliza cookies para rastrear a sessão do cliente.
  - Desvantagem: Não escala bem para ambientes distribuídos.
- **Tokens (JWT)**:
  - O token é autônomo e contém todas as informações necessárias (sem estado no servidor).
  - Utilizado em APIs e microserviços, facilitando a comunicação segura.
  - Melhor escalabilidade para sistemas distribuídos.

JSON Web Tokens (JWT): O que são?

- O JWT é um padrão aberto (RFC 7519) para transmitir informações entre partes de forma segura.
- Dividido em três partes:
  - **Header**: Define o tipo do token e o algoritmo de assinatura.
  - **Payload**: Contém as declarações (claims), como ‘sub‘ (identidade do usuário) e ‘exp‘ (expiração).
  - **Signature**: Garante a integridade do token, usando um segredo compartilhado ou chave privada.
- Tokens são assinados, mas não criptografados: as informações são legíveis, mas seguras contra modificações.

Vantagens do JWT em Sistemas Distribuídos

- **Sem Estado (Stateless)**: Não requer armazenamento de sessão no servidor.
- **Escalabilidade**: Facilita a autenticação em sistemas distribuídos e microserviços.
- **Portabilidade**: O token pode ser usado em várias camadas de aplicação, do frontend ao backend.
- **Autossuficiente**: O JWT carrega todas as informações de autorização no próprio token, eliminando a necessidade de consultas constantes ao banco de dados.

Uso do JWT para Autenticação e Autorização

- **Autenticação**:
  - O cliente envia suas credenciais para o servidor.
  - Se válidas, o servidor retorna um JWT, que é armazenado no cliente (geralmente no local storage ou em cookies).
- **Autorização**:
  - Em cada requisição subsequente, o cliente envia o JWT no cabeçalho de ‘Authorization‘.
  - O servidor verifica a assinatura e o payload do JWT para permitir ou negar o acesso ao recurso solicitado.

Armazenamento Seguro de Senhas

- Senhas nunca devem ser armazenadas em texto simples.
- Técnicas de armazenamento seguro:
  - **Hashing**: Usar uma função de hashing criptográfica (ex: bcrypt, Argon2).
  - **Salting**: Adicionar um valor aleatório (salt) a cada senha antes de aplicar o hash, para evitar ataques de rainbow table.
- O hashing deve ser "lento", dificultando ataques de força bruta.

Hashing vs. Encryption

- **Hashing**: Função unidirecional. Transformação que não pode ser revertida.
- **Encryption (Criptografia)**: Função bidirecional, com possibilidade de descriptografia usando uma chave.
- Para senhas, o **hashing** é mais seguro, pois não permite recuperação da senha original, mesmo com acesso ao banco de dados.

Protegendo APIs com JWT

- JWT é amplamente utilizado para proteger APIs RESTful.
- O token deve ser enviado no cabeçalho HTTP ‘Authorization‘ com o esquema ‘Bearer‘.
- Cada requisição a uma API protegida deve validar o token:
  - Verificar a assinatura do token.
  - Validar as declarações de tempo (‘exp‘, ‘iat‘).
- O backend deve rejeitar tokens inválidos ou expirados.

Práticas Seguras em APIs

- Usar **HTTPS** para garantir que os tokens não sejam interceptados por ataques man-in-the-middle.
- Implementar **rate limiting** para evitar ataques de força bruta.
- Cuidado com o **armazenamento de tokens** no frontend:
  - Preferir armazenamento em cookies com ‘HttpOnly‘ e ‘Secure‘ flags.
  - Evitar armazenar tokens no ‘localStorage‘ se possível (susceptível a ataques XSS).

Containers para Autenticação Segura

- Containers permitem a criação de ambientes isolados e replicáveis, facilitando o gerenciamento de autenticação e autorização.
- Vantagens:
  - Isolamento de serviços críticos (ex: autenticação).
  - Facilidade de replicação em ambientes distribuídos.
  - Infraestrutura como código: Pode-se versionar e gerenciar o ambiente inteiro (ex: Docker Compose).
- Exemplo de uso:
  - Um container separado para o MongoDB (armazenando usuários).
  - Um container para o backend Flask gerando e verificando JWTs.