Semana 16 - Aula 1

Tópico Principal da Aula: Configuração e Segurança de APIs

Subtítulo/Tema Específico: Implementação de Cross-Origin Resource Sharing (CORS)

Código da aula: [SIS]ANO2C2B3S16A1

Objetivos da Aula:

- Configurar e implementar cross-origin resource sharing (CORS) em APIs. [1]
- Desempenhar a cooperação ao trabalhar em equipe para resolver problemas de CORS. [2]

Recursos Adicionais:

- Recurso audiovisual para exibição de vídeos e imagens;
- Caneta e caderno para anotações;
- Laboratório de informática.
- Computador com internet.

Exposição do Conteúdo:

Referência do Slide: Slide 02 - Implementação de cross-origin resource sharing (CORS)

- **Definição:** CORS (Cross-Origin Resource Sharing) **é um mecanismo de segurança implementado nos navegadores web que controla como os recursos de uma página web podem ser requisitados por outro domínio que não seja o de onde o recurso se originou**. Em termos simples, ele permite que um navegador solicite recursos (como imagens, scripts, CSS, ou dados de API) de um servidor em um domínio diferente do domínio da página que fez a solicitação.
- **Aprofundamento/Complemento:** Sem o CORS, o navegador impõe uma política de mesma origem (Same-Origin Policy - SOP), que impede que scripts carregados de uma origem interajam com recursos de outra origem. Essa política é crucial para a segurança, pois evita que sites maliciosos leiam dados confidenciais de outros sites. No entanto, em cenários legítimos, como APIs públicas ou aplicações web que consomem serviços de diferentes domínios, o CORS se faz necessário para flexibilizar essa restrição de forma controlada. O CORS funciona adicionando cabeçalhos HTTP que informam ao navegador se a origem da requisição tem permissão para acessar os recursos.
- **Exemplo Prático: Imagine que você está desenvolvendo um aplicativo front-end em app.meudominio.com e ele precisa buscar dados de uma API hospedada em api.servicoexterno.com. Por padrão, o navegador bloquearia essa solicitação devido à política de mesma origem. Para permitir que seu aplicativo acesse a API, o servidor api.servicoexterno.com precisa enviar cabeçalhos CORS específicos na resposta, indicando que app.meudominio.com é uma origem permitida.**

- **Links de Vídeos:**
- **CORS na Prática Código fonte tv**
    - CORS (Cross-Origin Resource Sharing em 6 minutos) // Dicionário do …

Referência do Slide: Slide 03 - Fundamentos de servidores e segurança; Configuração e segurança de APIs

- **Definição:** **A segurança em servidores web e APIs é um aspecto crítico do desenvolvimento de sistemas, abrangendo desde a configuração básica de servidores até a implementação de mecanismos de proteção em APIs.** O objetivo é garantir que as aplicações sejam robustas contra ataques e que os dados sensíveis sejam protegidos.
- **Aprofundamento/Complemento:** **A configuração segura de APIs envolve a implementação de controles de acesso, criptografia, validação de entradas, tratamento de erros e a correta configuração de cabeçalhos de segurança, como os relacionados ao CORS**. Servidores web, por sua vez, devem ser configurados com as melhores práticas de segurança, como desabilitar serviços desnecessários, manter o software atualizado e implementar firewalls.
- **Exemplo Prático:** Para proteger um servidor web, pode-se desabilitar diretórios de listagem, restringir o acesso a arquivos de configuração sensíveis e usar certificados SSL/TLS para criptografar o tráfego. No contexto de APIs, além do CORS, é fundamental implementar autenticação (como OAuth 2.0 ou JWT) e autorização para garantir que apenas usuários e aplicações autorizados possam acessar recursos específicos.

Referência do Slide: Slide 10 - Qual é a importância do CORS para a segurança de APIs públicas?

- **Definição:** **O CORS é de suma importância para a segurança de APIs públicas, pois protege contra acessos não autorizados de origens diferentes, prevenindo ataques como Cross-Site Request Forgery (CSRF) e garantindo que apenas origens confiáveis possam interagir com os recursos da API**.
- **Aprofundamento/Complemento:** Embora a política de mesma origem (SOP) impeça que sites maliciosos leiam dados de outras origens, ela não impede que eles enviem requisições. Ataques CSRF exploram essa falha, induzindo o navegador da vítima a fazer uma requisição não intencional a um site confiável. O CORS atua como uma camada de segurança adicional ao exigir que o servidor API explicitamente conceda permissão a origens específicas, tornando mais difícil para um invasor explorar essa vulnerabilidade.
- **Exemplo Prático:** Se um atacante tentar enviar uma requisição maliciosa de site-malicioso.com para uma API em minha-api.com, e minha-api.com estiver configurada com CORS para permitir apenas requisições de meu-aplicativo.com, a requisição do site-malicioso.com será bloqueada pelo navegador do usuário.

Referência do Slide: Slide 10 - Quais são os principais cabeçalhos utilizados para configurar o CORS?

- **Definição:** Os principais cabeçalhos utilizados para configurar o CORS são: Access-Control-Allow-Origin, Access-Control-Allow-Methods, Access-Control-Allow-Headers, Access-Control-Allow-Credentials, Access-Control-Expose-Headers e Access-Control-Max-Age. [10]
- **Aprofundamento/Complemento:**
  - Access-Control-Allow-Origin: Essencial, define quais origens (domínios) podem acessar o recurso. Pode ser um domínio específico (https://example.com) ou um curinga (*) para permitir qualquer origem (não recomendado para APIs sensíveis).
  - Access-Control-Allow-Methods: Especifica os métodos HTTP (GET, POST, PUT, DELETE, OPTIONS, etc.) que são permitidos ao acessar o recurso.
  - Access-Control-Allow-Headers: Lista os cabeçalhos HTTP que podem ser usados na requisição.
  - Access-Control-Allow-Credentials: Indica se as credenciais (como cookies ou cabeçalhos de autorização HTTP) podem ser incluídas na requisição cross-origin.
  - Access-Control-Expose-Headers: Permite que o navegador acesse cabeçalhos da resposta que, por padrão, não seriam expostos a scripts no cliente.
  - Access-Control-Max-Age: Define por quanto tempo (em segundos) os resultados de uma requisição de pré-voo (preflight request) podem ser armazenados em cache. Requisições de pré-voo são requisições HTTP OPTIONS que o navegador envia para verificar se o servidor permite a requisição real. [16]
- **Exemplo Prático: Para permitir que uma aplicação React em http://localhost:3000 faça requisições GET e POST para uma API, o servidor da API retornaria nos cabeçalhos:**
  Access-Control-Allow-Origin: http://localhost:3000
  Access-Control-Allow-Methods: GET, POST
  Access-Control-Max-Age: 86400
-

Referência do Slide: Slide 11 - Como o CORS pode ajudar a controlar o acesso de diferentes origens à sua API?

- **Definição:** O CORS controla o acesso de diferentes origens à sua API, permitindo que você restrinja o acesso a origens confiáveis, métodos HTTP específicos e cabeçalhos autorizados, o que ajuda a evitar interações indesejadas com a API.
- **Aprofundamento/Complemento:** Ao configurar o CORS, você pode criar uma "lista branca" de domínios permitidos, garantindo que apenas aplicações autorizadas possam consumir sua API. Isso é particularmente importante para APIs que lidam com dados sensíveis ou funcionalidades críticas. A granularidade dos cabeçalhos CORS permite um controle preciso sobre o tipo de requisição e as informações que podem ser trocadas.

- **Exemplo Prático: Uma API de banco pode configurar CORS para permitir acesso apenas do domínio oficial do banco (banco.com), impedindo que outros sites tentem realizar operações em nome dos usuários. Se o cabeçalho Access-Control-Allow-Origin for configurado para banco.com, qualquer requisição de site-falso.com para a API será bloqueada pelo navegador.**

Referência do Slide: Slide 11 - Quais são os riscos de configurar o CORS de forma incorreta?

- **Definição:** A configuração incorreta do CORS pode levar a sérios riscos de segurança, incluindo acesso não autorizado, exposição de dados, credenciais vulneráveis e uma falsa sensação de segurança.
- **Aprofundamento/Complemento:**
  - **Acesso Não Autorizado:** Usar `Access-Control-Allow-Origin: *` com `Access-Control-Allow-Credentials: true` é uma vulnerabilidade grave. Isso expõe a API para qualquer origem, permitindo que sites maliciosos enviem requisições com credenciais do usuário.
  - **Exposição de Dados:** Configurar cabeçalhos ou métodos de forma insegura pode expor dados sensíveis ou permitir operações não intencionais. Por exemplo, permitir métodos HTTP desnecessários para um determinado endpoint.
  - **Credenciais Vulneráveis:** Permitir o envio de credenciais com um curinga (`*`) no cabeçalho de origem (`Access-Control-Allow-Origin`) é extremamente perigoso, pois um site malicioso pode enganar o navegador do usuário para enviar seus cookies ou tokens de autenticação para a API.
  - **Falsa Segurança:** Uma configuração inadequada pode criar a ilusão de segurança, enquanto na verdade deixa brechas significativas que podem ser exploradas por atacantes.
- **Exemplo Prático:** Se uma API que lida com informações de cartão de crédito permitir `Access-Control-Allow-Origin: *` e `Access-Control-Allow-Credentials: true`, um site de phishing poderia enviar uma requisição para essa API com os cookies de sessão do usuário, acessando suas informações financeiras sem o consentimento do usuário.

Semana 16 - Aula 2

Tópico Principal da Aula: Configuração e Segurança de APIs

Subtítulo/Tema Específico: Segurança em autenticação e autorização

Código da aula: [SIS]ANO2C2B3S16A2

Objetivos da Aula:

- Implementar práticas de segurança em processos de autenticação e autorização.

Recursos Adicionais:

- Recurso audiovisual para exibição de vídeos e imagens;
- Caneta e caderno para anotações;
- Laboratório de informática.
- Computador com internet.

Exposição do Conteúdo:

Referência do Slide: Slide 02 - Segurança em autenticação e autorização

- **Definição:** A segurança em autenticação e autorização é fundamental para proteger sistemas e APIs. A autenticação verifica a identidade do usuário (quem é você?), enquanto a autorização determina quais ações um usuário autenticado pode realizar (o que você pode fazer?).
- **Aprofundamento/Complemento:** Autenticação geralmente envolve a validação de credenciais (usuário e senha, tokens, biometria). Autorização, por outro lado, verifica as permissões do usuário para acessar recursos ou executar funções específicas após a autenticação. A combinação eficaz de ambos garante que apenas usuários legítimos e com as permissões corretas possam interagir com o sistema.
- **Exemplo Prático:** Em um sistema bancário, a autenticação ocorre quando você insere seu login e senha para acessar sua conta. A autorização entra em jogo quando o sistema verifica se você tem permissão para realizar uma transferência bancária ou acessar o extrato de outra conta.

Referência do Slide: Slide 07 - Entendemos como o protocolo OAuth 2.0 garante acesso seguro aos recursos do sistema sem expor credenciais sensíveis;

- **Definição:** O protocolo OAuth 2.0 é um padrão aberto para autorização que permite que um aplicativo obtenha acesso limitado a uma conta de usuário em um serviço HTTP (como Google, Facebook, etc.) sem que o aplicativo precise saber as credenciais do usuário. Ele garante acesso seguro aos recursos do sistema sem expor credenciais sensíveis.
- **Aprofundamento/Complemento:** O OAuth 2.0 define "fluxos de concessão" (grant flows) para diferentes cenários de uso. O fluxo "Authorization Code Flow" é o mais seguro e recomendado para aplicações web. Ele envolve o redirecionamento do usuário para o servidor de autorização (onde ele insere suas credenciais), que então retorna um código para o aplicativo cliente. O aplicativo cliente troca esse código por um token de acesso diretamente com o servidor de autorização, sem nunca ter acesso às credenciais do usuário.
- **Exemplo Prático:** Quando você clica em "Entrar com o Google" em um site, você está usando OAuth 2.0. O site não pede seu nome de usuário e senha do Google; em vez disso, ele o redireciona para o Google, você autoriza o acesso, e o Google envia um token de volta para o site, permitindo que ele acesse informações específicas do seu perfil no Google (como seu nome ou e-mail) sem que o site veja suas credenciais.

Referência do Slide: Slide 07 - Aprendemos sobre a estrutura e as vantagens do JSON Web Token (JWT) para autenticação baseada em tokens;

- **Definição:** JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma forma compacta e segura para a transmissão de informações entre partes como um objeto JSON. Ele é frequentemente usado para autenticação baseada em tokens.
- **Aprofundamento/Complemento:** Um JWT é composto por três partes, separadas por pontos: `header.payload.signature`.
    - **Header:** Contém o tipo do token (JWT) e o algoritmo de assinatura usado (ex: HMAC SHA256 ou RSA).
    - **Payload:** Contém as "claims" (declarações), que são declarações sobre uma entidade (geralmente o usuário) e metadados adicionais. Exemplos de claims incluem `iss` (emissor), `exp` (tempo de expiração) e `sub` (identificador do sujeito). [31]
    - Signature: É criada usando o header codificado, o payload codificado, um segredo (ou chave privada) e o algoritmo especificado no header. A assinatura garante que o token não foi alterado.
      As vantagens do JWT incluem ser compacto (pode ser enviado em URLs ou cabeçalhos HTTP), autossuficiente (contém todas as informações necessárias sobre o usuário) e seguro (devido à assinatura criptográfica).
- **Exemplo Prático:** Após um usuário fazer login, o servidor gera um JWT que contém o ID do usuário, seu papel (ex: administrador, cliente) e um tempo de expiração. Este token é enviado ao cliente, que o armazena e o envia em cada requisição subsequente no cabeçalho `Authorization`. O servidor pode então verificar a assinatura do token e usar as informações do payload para autenticar e autorizar a requisição.
- **Links de Vídeos:**
    - O que é JWT (JSON Web Token)? | Rocketseat

Referência do Slide: Slide 07 - Exploramos práticas de segurança, como expiração de tokens e proteção de endpoints sensíveis, para aumentar a confiabilidade das APIs.

- **Definição:** Para aumentar a confiabilidade e segurança das APIs, é crucial implementar práticas como a expiração de tokens e a proteção rigorosa de endpoints sensíveis.
- **Aprofundamento/Complemento:**
    - **Expiração de Tokens:** Limitar o tempo de vida de um token JWT (usando o claim `exp`) reduz a janela de oportunidade para um atacante caso o token seja comprometido. Tokens expirados devem ser invalidados e o usuário deve ser forçado a reautenticar.
    - **Proteção de Endpoints Sensíveis:** Endpoints que lidam com operações financeiras, dados pessoais ou configurações críticas devem ser protegidos com camadas adicionais de segurança, como verificação rigorosa de permissões, rate limiting para prevenir ataques de força bruta, e validação de entrada para evitar injeções.
- **Exemplo Prático:** Um endpoint de `GET /usuarios/{id}/transacoes` que retorna o histórico de transações de um usuário deve verificar se o usuário autenticado tem permissão para ver *apenas* suas próprias transações, e não as de outros usuários.

Além disso, o token JWT usado para acessar este endpoint deve ter um tempo de expiração curto para minimizar o risco em caso de roubo.

- **Links de Vídeos:**
  - Como Proteger Sua API: Dicas Essenciais de Segurança | AWS Brasil

Referência do Slide: Slide 10 - Explique como o protocolo OAuth 2.0 pode ser implementado para autenticar os usuários no sistema bancário.

- **Definição:** O OAuth 2.0 pode ser implementado para autenticar usuários em um sistema bancário utilizando o authorization code flow. Este fluxo é adequado para aplicações seguras, pois não expõe as credenciais do usuário ao cliente.
- **Aprofundamento/Complemento:**
  1. **Redirecionamento:** O cliente (aplicativo do usuário) redireciona o usuário para a página de autorização do servidor OAuth (servidor do banco).
  2. **Autenticação e Autorização:** O usuário autentica suas credenciais diretamente com o servidor OAuth. Após a autenticação e autorização, o servidor OAuth redireciona o cliente com um código de autorização.
  3. **Troca de Código por Token:** O cliente troca o código de autorização por um access token diretamente com o servidor OAuth. Esta troca é feita em um canal seguro (back-channel) e o código é de uso único.
  4. **Acesso a Recursos:** O access token é então utilizado para acessar os recursos protegidos da API do banco.
  5. Este processo garante que as credenciais do usuário (login/senha) sejam expostas apenas ao servidor OAuth do banco, e nunca ao aplicativo cliente.
- **Exemplo Prático:** Um aplicativo de gerenciamento financeiro de terceiros quer se integrar com o banco do usuário. Em vez de pedir as credenciais do banco, o aplicativo redireciona o usuário para a página de login do próprio banco. Após o login, o banco pergunta ao usuário se ele autoriza o aplicativo a acessar suas informações financeiras (ex: extrato). Se o usuário autorizar, o banco envia um código de autorização de volta para o aplicativo, que então o troca por um token de acesso para interagir com a API do banco.

Referência do Slide: Slide 10 - Quais informações são, geralmente, armazenadas no payload de um JWT e como elas podem ser utilizadas para autorização em APIs?

- **Definição:** O payload de um JWT geralmente contém "claims", que são declarações sobre a entidade (o usuário) e outros dados. Essas claims são utilizadas para autorizar acessos em APIs.
- **Aprofundamento/Complemento:**
  - **Claims Registrados:** São informações padronizadas, como iss (emissor do token), exp (tempo de expiração do token) e sub (identificador do usuário).

  - **Claims Públicos:** Dados definidos pela aplicação, como o papel do usuário (role) ou permissões específicas (scope).
  - **Claims Privados:** Informações específicas para o contexto da aplicação, que são acordadas entre as partes.

    - A assinatura do JWT (gerada com um segredo ou chave privada) garante a integridade do token, evitando manipulações. Apenas tokens válidos e assinados pelo servidor são aceitos. 43Para autorização, um `scope` pode, por exemplo, definir quais endpoints um usuário pode acessar.
- **Exemplo Prático:** Se o payload de um JWT contém a claim `role: "admin"`, a API pode verificar essa claim e conceder acesso a endpoints administrativos que um usuário comum (`role: "cliente"`) não poderia acessar.

Referência do Slide: Slide 11 - Cite um exemplo prático de como a API poderia proteger um endpoint sensível (como um de transações financeiras) usando JWT.

- **Definição:** Para proteger um endpoint sensível, como um de transações financeiras, usando JWT, a API deve validar o token em cada requisição, verificando a assinatura, o tempo de expiração e as permissões do usuário contidas no payload.
- **Aprofundamento/Complemento:**
    1. **Login e Emissão do JWT:** O usuário realiza o login, e o servidor OAuth (ou o próprio servidor da API) emite um JWT contendo claims como `sub` (identificador do usuário), `role` (papel do usuário, ex: "cliente"), e `exp` (tempo de expiração do token).
    2. **Envio do Token:** Ao acessar o endpoint `/transacoes`, o cliente envia o JWT no cabeçalho HTTP `Authorization: Bearer <token>`.
    3. **Validação na API:** A API valida o token verificando:
        - **Assinatura:** Garante que o token não foi adulterado.
        - **Tempo de Expiração:** Confirma que o token ainda é válido.
        - **Role/Permissões:** Verifica se o `role` no payload é adequado para acessar as transações financeiras. Por exemplo, apenas usuários com `role: "cliente"` podem acessar suas próprias transações.
        - Somente tokens válidos com as permissões apropriadas garantem acesso às transações.
- **Exemplo Prático:** Um usuário tenta acessar `/transacoes/123` para ver a transação de ID 123. Se o JWT enviado indicar que o `sub` (identificador do usuário) é `user_A`, e a transação 123 pertence a `user_B`, a API, após validar o token, negaria o acesso, mesmo que o token seja válido, porque o usuário não tem permissão para ver transações de outros.

Referência do Slide: Slide 11 - Quais medidas adicionais de segurança poderiam ser adotadas no sistema para complementar o uso de OAuth 2.0 e JWT?

- **Definição:** Além do OAuth 2.0 e JWT, diversas medidas adicionais de segurança podem ser adotadas para aumentar a robustez de um sistema, incluindo criptografia, políticas de CORS, validação de token, rotação de chaves, scope detalhado e lista de revogação.
- **Aprofundamento/Complemento:**
    - **Criptografia e HTTPS:** Todas as comunicações devem ser feitas exclusivamente por HTTPS para prevenir ataques man-in-the-middle e garantir a confidencialidade dos dados em trânsito.

    - **Políticas de CORS:** Configurar o servidor para permitir apenas domínios autorizados a acessar os recursos, prevenindo requisições de origens maliciosas.
    - **Validação de Token:** Utilizar um middleware na API para verificar rigorosamente a validade e expiração do JWT antes de processar qualquer requisição.
    - **Rotação de Chaves:** Atualizar regularmente as chaves privadas usadas para assinar os tokens JWT. Isso limita o impacto de uma chave comprometida.
    - **Scope Detalhado:** Limitar os privilégios concedidos pelos tokens (via `scope` no payload do JWT) seguindo o princípio do privilégio mínimo, ou seja, conceder apenas as permissões estritamente necessárias.
    - **Lista de Revogação:** Manter uma lista de tokens que foram revogados antecipadamente (por exemplo, em caso de logout ou comprometimento), garantindo que esses tokens não sejam mais aceitos, mesmo que não tenham expirado.
- **Exemplo Prático:** Implementar HTTPS é uma medida básica. Para um sistema bancário, além de HTTPS, o servidor pode ter uma política de CORS que permite apenas acesso do domínio do próprio banco. Se um token for roubado, o usuário pode "deslogar de todos os dispositivos", o que adicionaria o token à lista de revogação, impedindo seu uso futuro.
