**1 - Conceitos Iniciais**

**APIs**

* **API**: É um termo amplo que engloba qualquer interface que permita a interação entre diferentes softwares.
* **Web Service**: Um tipo de **API** que opera na web e utiliza protocolos como **SOAP** e **REST**, focando na interoperabilidade entre sistemas.
* **Web Service REST**: Um subtipo de **Web Service**, geralmente construído seguindo os princípios **REST** e utilizando o protocolo HTTP/HTTPS, mais leve e adequado para a web moderna.

**SOAP e REST**

* **REST** (REPRESENTATIONAL STATE TRANSFER): Não é um protocolo, mas um estilo arquitetural que geralmente utiliza o protocolo HTTP/HTTPS e define um conjunto de princípios para criar serviços web eficientes e escaláveis. Pode ser implementado sobre qualquer protocolo de comunicação
* **SOAP** (SIMPLE OBJECT ACESS PROTOCOL): É um protocolo de comunicação rigoroso e baseado em XML, utilizado para a troca de informações estruturadas em redes distribuídas, pode operar sobre vários de tipos de protocolos, tais como HTTP, SMTP e TCP.

**PRINCÍPIOS DO REST**

O estilo arquitetural REST (Representational State Transfer) é amplamente utilizado para implementar APIs Web devido à sua simplicidade e flexibilidade. REST é baseado em um conjunto de princípios que definem como os recursos devem ser acessados e manipulados através da Web. Aqui estão os principais aspectos do REST:

1. **Client-Server**: A arquitetura REST é baseada em uma separação clara entre cliente e servidor. O cliente faz requisições para o servidor, que processa e retorna as respostas, permitindo uma maior escalabilidade e modularidade.
2. **Stateless**: Cada requisição do cliente para o servidor deve conter todas as informações necessárias para que o servidor a entenda e a processe. O servidor não deve armazenar nenhum estado sobre o cliente entre as requisições, o que facilita a escalabilidade.
3. **Cacheable**: As respostas das requisições devem ser definidas como cacheáveis ou não-cacheáveis, para que o cliente possa armazenar respostas reutilizáveis e reduzir a carga no servidor.
4. **Uniform Interface**: A interface entre o cliente e o servidor deve ser uniforme, o que significa que deve haver uma forma padrão de interagir com os recursos. Isso é frequentemente realizado através de convenções de URL e métodos HTTP (GET, POST, PUT, DELETE).
5. **Layered System**: A arquitetura REST pode ser composta por várias camadas, cada uma com uma responsabilidade específica, como segurança, balanceamento de carga e caching, sem que o cliente precise conhecer a existência dessas camadas.
6. **Code on Demand (opcional)**: Servidores podem estender a funcionalidade dos clientes através da transferência de código executável, como scripts JavaScript. Esse princípio é opcional e nem todas as APIs REST o implementam.

**INTERFACE UNIFORME**

1. **Métodos HTTP Comuns**

* **GET**: Recupera representações de recursos.
* **POST**: Submete dados para serem processados, criando ou modificando recursos.
* **PUT**: Atualiza um recurso existente ou cria um se ele não existir.
* **DELETE**: Remove um recurso.
* **PATCH**: Aplica modificações parciais a um recurso.

1. **Identificação de Recursos**: Cada recurso é identificado de forma única através de um URL (Uniform Resource Locator). Por exemplo, um pedido específico pode ser identificado por um URL.
2. **Manipulação de Recursos através de Representações**: Recursos são manipulados através de representações, que são geralmente documentos JSON ou XML que descrevem o estado do recurso. Por exemplo, ao enviar uma representação JSON de um pedido ao servidor, você pode atualizar o estado desse pedido.
3. **Mensagens Auto descritivas**: Cada mensagem (requisição ou resposta) contém informações suficientes para descrever como ser processada. Isso inclui cabeçalhos de metadados que informam o formato dos dados, status de resposta HTTP etc.
4. **Hipermídia como o Motor do Estado da Aplicação (HATEOAS)**: Clientes interagem com a aplicação através de hipermídia fornecida pelo servidor. Isso significa que a resposta do servidor inclui links que o cliente pode seguir para descobrir recursos relacionados ou realizar ações adicionais.
5. **Responder apropriadamente:** O uso adequado de códigos de status HTTP é essencial para uma interface uniforme, pois permite que os clientes compreendam rapidamente o resultado de suas solicitações. Códigos como **200 (OK)**, **201 (Created)**, **400 (Bad Request)**, **404 (Not Found)**, entre outros, informam o estado da operação.

**HATEOAS e INTERFACE UNIFORME**

O HATEOAS é uma parte crítica do princípio da Interface Uniforme porque ele permite que o cliente navegue e interaja com a aplicação de forma dinâmica, sem a necessidade de conhecimento prévio da estrutura da API além do ponto inicial de entrada. Com HATEOAS, a API se torna auto descritiva e o cliente pode descobrir novas funcionalidades e recursos conforme navega pelas respostas fornecidas pelo servidor.

**VANTAGENS DO REST**

* **Simplicidade**: Facilidade de uso e entendimento por desenvolvedores.
* **Flexibilidade**: Adaptável a uma ampla variedade de aplicações.
* **Desempenho**: Eficiência na comunicação e no uso de recursos de rede.

**DESVANTAGENS DO REST**

* **Limitações em operações complexas**: Pode ser menos adequado para operações de longa duração ou muito complexas.
* **Sem estado**: A ausência de estado pode ser uma desvantagem em aplicações que exigem persistência entre requisições.

O REST continua a ser uma escolha popular para a criação de APIs Web devido à sua compatibilidade com os padrões da Web e à sua facilidade de integração com diversas tecnologias.

**2 – Mensagens HTTP**

**REQUISIÇÕES HTTP**

Servidor

Requisição

Resposta

Cliente





**VERBOS HTTP**



**HTTP – HTTPS e HSTS**

Os conceitos de HTTPS, SSL, TLS e HSTS estão interrelacionados e são essenciais para a segurança na comunicação na internet. Abaixo, segue uma explicação de cada um e como se conectam:

1. **HTTPS (HyperText Transfer Protocol Secure)**:

* HTTPS é a versão segura do HTTP, o protocolo usado para transferência de dados na web.
* Ele utiliza criptografia para proteger a comunicação entre o navegador do usuário e o servidor web.
* A segurança é provida pelo uso de SSL/TLS.

1. **SSL (Secure Sockets Layer)**:

* SSL é um protocolo de criptografia que foi inicialmente desenvolvido para assegurar a comunicação via internet.
* Ele cria um canal seguro sobre uma rede insegura.
* SSL foi amplamente utilizado, mas suas versões mais antigas têm vulnerabilidades conhecidas, por isso foi substituído pelo TLS.

1. **TLS (Transport Layer Security)**:

* TLS é a evolução do SSL e é atualmente o protocolo padrão para a criptografia de comunicações na web.
* Ele fornece três principais propriedades de segurança: confidencialidade (por meio da criptografia), integridade (assegurando que os dados não foram alterados durante a transferência) e autenticação (confirmando a identidade das partes envolvidas na comunicação).
* Versões mais modernas do TLS são recomendadas para garantir a segurança dos dados.

1. **HSTS (HTTP Strict Transport Security)**:

* HSTS é um mecanismo de segurança que força os navegadores a interagirem apenas com servidores usando conexões HTTPS.
* Quando um servidor envia um cabeçalho HSTS, ele informa ao navegador que todas as futuras interações devem ser realizadas usando HTTPS.
* Isso ajuda a prevenir ataques como o downgrade, onde um invasor força a conexão a usar HTTP não seguro, e ataques de sequestro de sessão.

**CENÁRIO REAL**

1. **Primeira Visita**:

* Um usuário acessa http://www.exemplo.com pela primeira vez. A solicitação é redirecionada para https://www.exemplo.com e o navegador recebe o cabeçalho HSTS.

1. **Armazenamento de Política HSTS**:

* O navegador armazena a política HSTS e agora sabe que deve acessar sempre https://www.exemplo.com para os próximos 31536000 segundos (1 ano).

1. **Visita Subsequente**:

* Em uma visita subsequente, o usuário digita http://www.exemplo.com ou clica em um link HTTP para www.exemplo.com. O navegador automaticamente converte a solicitação para https://www.exemplo.com sem enviar a solicitação HTTP.

1. **Garantia de Segurança Contínua**:

* Todas as comunicações futuras com www.exemplo.com são garantidas como seguras (via HTTPS), protegendo contra os ataques *man-in-the-middle* e garantindo a integridade dos dados transmitidos.

**3 – Implementação**

**INJEÇÃO DE DEPENDÊNCIA - DI**

A Injeção de Dependência é um padrão para implementar a **Inversão de Controle (IOC)** e, assim, reduzir o acoplamento entre os objetos. Ao aplicar a Injeção de Dependência, fazemos com que um objeto forneça as dependências do outro objeto.

**TEMPO DE VIDA DOS OBJETOS POR TIPO DE SERVIÇO**

1. **Transient:** uma nova instância é criada a cada chamada do serviço de injeção de dependência.
2. **Scoped**: em uma mesma requisição, injeções de dependência iguais (que usam a mesma classe), em qualquer lugar da aplicação, usarão a mesma instância.
3. **Singleton**: uma mesma instância é usada por todo tempo de vida da aplicação, para todas as injeções de dependência iguais (que usam a mesma classe).

**PASSOS PARA IMPLEMENTAÇÃO DA API (APROXIMAÇÃO CODE -FIRST)**

1. Criar as classes de domínio (Models), com os devidos relacionamentos de modelagem.
2. Adicionar dos pacotes Nuget: EntityFrameworkCore (SQLServer, Design e Tools). Para usar a linha de comando, instalar também Tools na linha de comando.
3. Criação da Classe de Contexto (que herda de DbContext), para fazer o mapeamento objeto-relacional.
4. Inserção da *string* de conexão em no arquivo *appsettings.json* dentro do objeto *ConnectionStrings*. Configurar a *string* de conexão e usar o resultado para criar o serviço no contêiner DI nativo.
5. Criar e aplicar as *migrations* (sincroniza as classes de domínio com o banco de dados, sem perder os dados existentes).
6. Adicionar os *Controllers*.