Aula 1

Created	@October 28, 2023 11:22 AM
Data	@January 31, 2024
Status	Iniciado
Tipo	Aula

Aplicações orientadas a serviços

Introdução a arquitetura de aplicações orientadas a serviços

A arquitetura de aplicações orientadas a serviços (SOA - Service-Oriented Architecture) é uma abordagem de design que se concentra na construção de sistemas modulares e interoperáveis, nos quais os componentes individuais são serviços independentes que se comunicam através de interfaces padronizadas. Aqui está uma introdução aos principais conceitos da SOA:

Princípios da SOA:

1. **Serviços:**
    - A SOA descompõe uma aplicação em serviços, que são unidades independentes e autônomas de funcionalidade de negócios.
    - Cada serviço representa uma peça específica de lógica de negócios e é projetado para ser reutilizável e interoperável.
2. **Interoperabilidade:**
    - Serviços são projetados para se comunicar de maneira padronizada, permitindo a interoperabilidade entre diferentes sistemas e tecnologias.
3. **Descoberta Dinâmica:**

- A SOA permite a descoberta dinâmica de serviços, facilitando a localização e a integração de novos serviços sem uma forte dependência estática.

4. **Padrões de Comunicação:**
   - A comunicação entre serviços segue padrões como SOAP (Simple Object Access Protocol), REST (Representational State Transfer) ou outros protocolos comuns.
5. **Reusabilidade:**
   - Os serviços são projetados para serem reutilizáveis em diferentes contextos, promovendo a eficiência e a consistência no desenvolvimento de software.

Componentes da Arquitetura SOA:

1. **Serviços:**
   - Unidades autônomas de funcionalidade de negócios, acessíveis via rede.
2. **Contratos:**
   - Definem as interfaces dos serviços, incluindo operações, parâmetros e formatos de dados.
3. **Descoberta de Serviços:**
   - Mecanismos que facilitam a localização dinâmica de serviços disponíveis na rede.
4. **Políticas e Governança:**
   - Estabelecem diretrizes para o desenvolvimento, implementação e evolução dos serviços.
5. **Middleware de Serviços:**
   - Fornece funcionalidades como segurança, transações e roteamento para os serviços.

Benefícios da SOA:

1. **Flexibilidade e Agilidade:**
   - Facilita a adaptação rápida a mudanças de requisitos de negócios.

2. **Reusabilidade:**
    - Encoraja o desenvolvimento de serviços reutilizáveis, reduzindo a redundância de código.
3. **Integração:**
    - Permite a integração eficiente de sistemas heterogêneos.
4. **Interoperabilidade:**
    - Facilita a comunicação entre diferentes tecnologias e plataformas.
5. **Escalabilidade:**
    - Os serviços podem ser escalados independentemente para lidar com demandas variáveis.

Desafios e Considerações:

1. **Complexidade:**
    - Gerenciar a complexidade de uma arquitetura SOA pode ser desafiador.
2. **Segurança:**
    - É necessário implementar medidas robustas de segurança para proteger a comunicação entre serviços.
3. **Governança:**
    - Uma governança sólida é crucial para garantir a consistência e a conformidade com padrões.
4. **Monitoramento:**
    - Monitorar o desempenho e a saúde dos serviços é essencial para garantir um ambiente operacional estável.

Qual a diferença entre Microservices e Webservices

1. **Definição:**
    - Web Services são um conjunto de padrões de comunicação que permitem a interação entre sistemas heterogêneos pela internet.

- Eles são serviços baseados em padrões como SOAP (Simple Object Access Protocol) ou REST (Representational State Transfer).

2. **Protocolos:**
   - Web Services podem usar diferentes protocolos de comunicação, como HTTP, SMTP, FTP, etc.
   - Exemplos incluem SOAP Web Services (usando XML) e RESTful Web Services (usando JSON ou XML).
3. **Granularidade:**
   - Web Services podem ter uma granularidade variável, desde serviços grandes e monolíticos até serviços mais granulares.
4. **Independência Tecnológica:**
   - Web Services permitem a comunicação entre diferentes tecnologias e plataformas.

Microservices:

1. **Definição:**
   - Microservices, ou microsserviços, são uma abordagem arquitetônica para o desenvolvimento de software, onde uma aplicação é dividida em serviços independentes e autônomos.
2. **Escopo:**
   - Cada microsserviço é uma unidade de desenvolvimento, implantação e escala independente, focada em uma única funcionalidade de negócios.
3. **Comunicação:**
   - Os microservices geralmente se comunicam por meio de protocolos leves como HTTP/REST ou mensagens assíncronas.
4. **Independência de Implantação:**
   - Cada microserviço é implantado separadamente, permitindo atualizações e escalabilidade independentes.
5. **Resiliência e Tolerância a Falhas:**

- A arquitetura de microsserviços é projetada para ser resiliente, com a capacidade de lidar com falhas em um serviço sem afetar outros.

6. **Desenvolvimento e Manutenção:**
    - Os microservices facilitam o desenvolvimento ágil e a manutenção contínua, pois cada serviço pode ser desenvolvido e atualizado separadamente.

O que é uma API (Application Programming Interface)?

Uma API (Interface de Programação de Aplicações) é um conjunto de regras e definições que permite a interação entre diferentes softwares. É uma ponte que permite que aplicativos se comuniquem entre si, permitindo que desenvolvedores acessem funcionalidades específicas de um software, serviço ou plataforma sem precisar entender a complexidade interna do sistema.

Principais Características de uma API:

1. **Padrões de Comunicação:**
    - As APIs definem os métodos e protocolos que os desenvolvedores podem utilizar para interagir com um serviço ou aplicativo.
2. **Abstração de Funcionalidades:**
    - As APIs fornecem uma camada de abstração, permitindo que os desenvolvedores usem funcionalidades específicas sem precisar entender a implementação interna.
3. **Reusabilidade:**
    - Uma API bem projetada é reutilizável, o que significa que ela pode ser utilizada em diferentes contextos e por diferentes desenvolvedores.
4. **Segurança:**
    - As APIs geralmente incluem mecanismos de segurança, como autenticação e autorização, para proteger o acesso não autorizado.
5. **Documentação:**
    - Boas APIs vêm com documentação detalhada, explicando como usá-las, quais são seus endpoints, métodos disponíveis, parâmetros aceitos, entre outros.

Tipos de APIs:

1. **APIs Web:**
    - Tornaram-se muito comuns, utilizando protocolos HTTP/HTTPS para comunicação. APIs RESTful e SOAP são exemplos populares.
2. **Bibliotecas e Frameworks:**
    - Muitas linguagens de programação fornecem bibliotecas e frameworks que são essencialmente APIs para facilitar o desenvolvimento de software.
3. **Sistemas Operacionais:**
    - APIs de sistemas operacionais fornecem funcionalidades para interação com hardware, gerenciamento de arquivos, redes, etc.
4. **Bancos de Dados:**
    - Bancos de dados normalmente têm APIs para que aplicativos possam acessar e manipular dados armazenados.
5. **Redes Sociais:**
    - Plataformas como Facebook, Twitter e Google fornecem APIs para que desenvolvedores possam integrar suas aplicações com essas redes.

Exemplo Prático:

Suponha que você esteja desenvolvendo um aplicativo móvel que precise acessar informações de previsão do tempo. Em vez de criar toda a lógica para coletar esses dados, você pode usar uma API de serviços meteorológicos. Essa API fornecerá métodos que você pode chamar para obter informações precisas de previsão do tempo sem precisar entender como os dados são coletados e processados internamente pela API.

Em resumo, APIs desempenham um papel fundamental na conectividade e na integração de sistemas, permitindo que desenvolvedores aproveitem funcionalidades existentes e criem aplicativos mais eficientes e poderosos.

Principais protocolos utilizados (HTTP, HTTPS, MQTT, REST, SOAP);

1. HTTP (Hypertext Transfer Protocol):

- **Descrição:** Protocolo baseado em texto usado para a transferência de dados na World Wide Web. É a base para a comunicação na web e é fundamental para a transferência de documentos hipermídia, como páginas HTML.
- **Utilização:** Amplamente utilizado em aplicações web e na construção de APIs RESTful.

2. HTTPS (Hypertext Transfer Protocol Secure):

- **Descrição:** Similar ao HTTP, mas com uma camada adicional de segurança fornecida pelo protocolo TLS (Transport Layer Security) ou pelo seu antecessor, o SSL (Secure Sockets Layer). A comunicação é criptografada para maior segurança.
- **Utilização:** Essencial para comunicações seguras na web, como transações financeiras online e autenticação.

3. MQTT (Message Queuing Telemetry Transport):

- **Descrição:** Protocolo leve de mensagens e publicação/assinatura, projetado para dispositivos com restrições de largura de banda e recursos. É utilizado em ambientes IoT (Internet of Things) e para comunicação entre dispositivos.
- **Utilização:** Comum em aplicações IoT para troca de mensagens entre dispositivos e servidores.

4. REST (Representational State Transfer):

- **Descrição:** Um estilo arquitetural para projetar serviços web que utiliza os métodos HTTP padrão (GET, POST, PUT, DELETE) para operações CRUD (Create, Read, Update, Delete). É baseado no conceito de recursos e é conhecido por ser simples e escalável.
- **Utilização:** Amplamente utilizado em serviços web para construir APIs (APIs RESTful) devido à sua simplicidade e flexibilidade.

5. SOAP (Simple Object Access Protocol):

- **Descrição:** Protocolo de comunicação baseado em XML usado para trocar mensagens estruturadas em ambientes distribuídos. Define um conjunto de regras para estruturar mensagens e métodos para comunicação entre aplicações.
- **Utilização:** Mais comum em serviços web mais antigos, onde a interoperabilidade é crucial, mas pode ser considerado mais complexo em comparação com REST.

Esses protocolos são escolhidos com base nos requisitos específicos da aplicação, nas características desejadas (como segurança e eficiência), e no contexto em que estão sendo aplicados. Cada um tem suas próprias vantagens e casos de uso ideais, e a escolha dependerá das necessidades específicas do projeto.

Introdução a RESTFUL e modelagem baseada em recursos

RESTful é um estilo de arquitetura de software que é frequentemente utilizado na construção de serviços web. Ele se baseia nos princípios do REST, introduzidos por Roy Fielding em sua tese de doutorado. REST é um conjunto de diretrizes que, quando seguidas, facilitam a criação de sistemas web eficientes, escaláveis e interconectados.

Princípios Básicos do RESTful:

1. **Recursos (Resources):**
    - Tudo em um sistema RESTful é considerado um recurso, que pode ser um objeto, serviço ou qualquer outra entidade.
    - Exemplos de recursos podem incluir usuários, produtos, postagens de blog, etc.
2. **Identificadores Únicos (URI - Uniform Resource Identifier):**
    - Cada recurso é identificado por um URI exclusivo, que é usado para acessar e manipular esse recurso.
3. **Representação:**
    - Um recurso pode ter diferentes representações, como JSON, XML, HTML, entre outras.
    - A representação é enviada ou recebida em uma requisição HTTP.

4. **Estado (Stateless):**
    - A comunicação é sem estado, o que significa que cada requisição do cliente para o servidor deve conter todas as informações necessárias para entender e processar a requisição.
    - A sessão do cliente é mantida no cliente, não no servidor.
5. **Operações Padrão HTTP:**
    - Utilização dos métodos HTTP padrão (GET, POST, PUT, DELETE) para realizar operações em recursos.
    - Cada método tem um significado específico: GET (obter dados), POST (criar um novo recurso), PUT (atualizar um recurso existente), DELETE (remover um recurso).

Modelagem Baseada em Recursos:

A modelagem baseada em recursos é um conceito-chave na arquitetura RESTful. Isso significa que, ao projetar uma API RESTful, você deve pensar em termos de recursos e como eles podem ser manipulados usando os métodos HTTP. Alguns princípios importantes incluem:

1. **Identificação dos Recursos:**
    - Identifique os recursos que sua aplicação gerencia. Por exemplo, em um sistema de blog, os recursos podem ser postagens, comentários, usuários, etc.
2. **Design da URI:**
    - Escolha URIs significativos e intuitivos para identificar cada recurso.
    - Por exemplo: `/users` para listar todos os usuários ou `/users/{id}` para um usuário específico.
3. **Métodos HTTP:**
    - Utilize os métodos HTTP de acordo com suas operações semânticas. Por exemplo, use GET para recuperar um recurso, POST para criar um novo recurso, PUT para atualizar um recurso existente e DELETE para remover um recurso.
4. **Representações:**

- Considere as diferentes representações que seu recurso pode ter (JSON, XML) e permita que o cliente especifique a preferência de representação.

5. **Relacionamentos entre Recursos:**
    - Se houver relacionamentos entre recursos, projete a API de modo que seja fácil navegar entre eles usando URIs.

Ao seguir esses princípios, você pode criar APIs RESTful que são intuitivas, fáceis de entender e que seguem as melhores práticas de design. Isso facilita a interoperabilidade e a escalabilidade das suas aplicações.

Como construir aplicações orientadas a serviços

1. Identificar Serviços:

- Decomponha a aplicação em funções ou serviços lógicos independentes.
- Identifique os limites de contexto e a responsabilidade de cada serviço.

2. Padrões de Comunicação:

- Utilize protocolos padrão para comunicação entre serviços, como REST, SOAP ou gRPC.
- Implemente contratos claros e documentados para os serviços.

3. Gerenciamento de Dados:

- Separe a lógica de negócios dos dados, utilizando serviços dedicados para manipulação de dados.
- Considere o uso de bases de dados especializadas para cada serviço.

4. Segurança:

- Implemente mecanismos de segurança robustos para autenticação e autorização.
- Considere a utilização de tokens JWT (JSON Web Tokens) para autenticação.

5. Descoberta de Serviços:

- Utilize um mecanismo de descoberta de serviços para facilitar a localização dinâmica de serviços disponíveis.
- Considere o uso de ferramentas como Consul, Eureka ou etcd.

6. Escalabilidade:

- Projete cada serviço para ser escalável independentemente dos outros.
- Utilize técnicas de escalabilidade horizontal para lidar com aumentos na carga.

7. Monitoramento e Logging:

- Implemente registros (logs) e monitoramento para cada serviço.
- Utilize ferramentas de monitoramento centralizado para facilitar a detecção e resolução de problemas.

8. Resiliência:

- Implemente práticas de resiliência, como retries automáticos e circuit breakers, para lidar com falhas temporárias.
- Considere a implementação de filas de mensagens para garantir a entrega de mensagens assíncronas.

9. Gerenciamento de Transações:

- Considere o uso de transações distribuídas ou modelos de consistência eventual para manter a integridade dos dados entre serviços.

10. Documentação:

- Mantenha documentação clara e atualizada para cada serviço.
- Forneça informações sobre como os serviços se comunicam e as dependências entre eles.

11. Testes Automatizados:

- Implemente testes automatizados abrangentes para cada serviço, incluindo testes unitários, de integração e de sistema.

12. Cultura DevOps:

- Promova uma cultura de colaboração entre desenvolvimento e operações para facilitar a entrega contínua e a manutenção eficiente.

13. Governança:

- Estabeleça políticas de governança para garantir a consistência na implementação e evolução dos serviços.
- Monitore regularmente a conformidade com essas políticas.

Prático:

- Pesquisar um exemplo real de webservice
- Pesquisar um exemplo real de micro serviço
- Pesquisar sobre boas práticas na implementação de uma api RESTFUL