Desafio VR Desenvolvimento

1. Recebemos um código desenvolvido por terceiros de um sistema que possui alto volume de lógica de negócio e apresenta as seguintes características:

   - O sistema recebe requisições REST, está dividido em camadas e possui classes de domínio;
   - O controller recebe a requisição e está com toda lógica de negócio. Monta e repassa o domínio para a aplicação;
   - A aplicação tem a responsabilidade de repassar o objeto pronto para o repositório;
   - O repositório apenas persiste os objetos mapeados do hibernate através de spring data;
   - O domínio apenas faz o mapeamento para o BD;
   - Nenhum teste unitário foi escrito.
   - O sistema está escrito em java para rodar como spring boot.

   Apresenta observações/problemas sobre essa solução.
   Comente qual(is) a(s) sua(s) estratégia(s) para melhorar este sistema em termos de qualidade e manutenção. Justifique suas decisões.

   Minhas observações para o sistema são:

   1. A priori não existe uma documentação a API para clientes que desejam integrar
   2. Não existe monitoramento da aplicação
   3. Não existe ferramentas de build e qualidade de código integrada
   4. O controller está com toda lógica de negócio, o que viola o princípio de responsabilidade única do SOLID e pode dificultar a manutenção e evolução do código.
   5. A aplicação tem a responsabilidade de repassar o objeto pronto para o repositório, o que pode tornar difícil a implementação de regras de negócio que envolvem múltiplas entidades e violam o princípio de inversão de dependência do SOLID.
   6. Não há testes unitários, o que pode levar a problemas de regressão e tornar a evolução do código mais difícil.
   7. A arquitetura não segue os princípios de Clean Architecture, onde as camadas externas dependem das camadas internas e não o contrário.

   Para melhorar o código, podemos seguir diversas estratégias. Segue abaixo algumas estratégias para melhorar este sistema em termos de qualidade e manutenção, de acordo com os princípios do SOLID, Clean Code e Clean Architecture:

   1. Podemos implementar um monitoramento da aplicação com Actuator, Health, Info, e atté utilizar Grafana/Prometheus, Dynatrace… para visualização da informaçõe de observabildiade.
   2. Podemos utilizar um ferramenta de o conceito de API-First para documentar as apis, temos ferramentas como o Stoplight ou Swagger.
   3. Separar a lógica de negócio do controller e criar classes de serviço. Isso segue o princípio de responsabilidade única do SOLID, onde cada classe tem uma única responsabilidade. Além disso, permite que a lógica de negócio seja reutilizada em outros controllers ou em outros pontos do sistema.
   4. Aplicar o princípio de inversão de dependência do SOLID, invertendo a dependência da aplicação para o repositório. Isso pode ser feito através da injeção de dependência e do uso de interfaces para representar o contrato que o repositório deve seguir. Isso torna o código mais flexível e adaptável a mudanças.
   5. Escrever testes unitários para cada camada do sistema. Isso segue o princípio de responsabilidade única do SOLID e ajuda a garantir que cada camada está funcionando corretamente. Além disso, a escrita de testes unitários ajuda a detectar problemas de regressão e torna a evolução do código mais segura.
   6. Adotar a arquitetura em camadas de Clean Architecture, onde as camadas externas dependem das camadas internas e não o contrário. Isso torna o sistema mais flexível, permitindo que as camadas internas sejam facilmente substituídas ou alteradas sem afetar as camadas externas. Além disso, essa abordagem ajuda a manter a separação de responsabilidades entre as camadas.
   7. Podemos uma ferramenta de migração de banco de dados de código aberto, como o Flyway
   8. Podemos utilizar conteinerização, como docker e kubernetes para facilitar a implantação em nuvem.

2. Descreva quais são as principais limitações ao se adotar servidores de aplicação em uma arquitetura orientada a microsserviços.

Quando estamos em uma arquitetura orientada a microserviços, teremos vários nós de microserviços com responsabilidades únicas executando em diferentes pontos da rede. Dessa forma, seriam necessários vários Servidores de Aplicação para executar os leves nós de microserviços, a qual consumiram muitos recursos de hardware desnecessariamente. Pois Servidores de aplicação foram pensados para rodar grandes aplicações, como as aplicações monolíticas que são muito robustas. Nós leves de um microserviço não precisam de todos os recursos disponibilizados em um servidor de aplicação.

A adoção de servidores de aplicação em uma arquitetura orientada a microsserviços pode apresentar algumas limitações, como:

1. Alto acoplamento: servidores de aplicação geralmente oferecem uma plataforma de execução para várias aplicações, o que pode resultar em alto acoplamento entre as aplicações hospedadas no mesmo servidor, tornando difícil a evolução e manutenção independentes de cada microsserviço.
2. Grande footprint: servidores de aplicação podem ter um grande footprint em termos de recursos de hardware e software, o que pode ser um problema em ambientes com recursos limitados. Podemos comparar com máquinas virtuais e contâiners docker, no qual a máquina virtual consome mais recursos de harware e sofware do que um contâiner.
3. Dificuldade em escalar microsserviços individualmente: servidores de aplicação podem limitar a capacidade de escalar microsserviços individualmente, já que geralmente requerem que toda a aplicação seja escalada em conjunto.
4. Limitações de linguagem: alguns servidores de aplicação podem ter limitações em termos de suporte a linguagens de programação, o que pode limitar a escolha de tecnologias para a implementação dos microsserviços.
5. Complexidade de configuração: servidores de aplicação geralmente possuem uma grande quantidade de configurações e opções de configuração, o que pode tornar a configuração e o gerenciamento do ambiente complexos.

Para superar essas limitações, pode ser mais adequado utilizar outras tecnologias e ferramentas que permitam uma maior independência e escalabilidade dos microsserviços, como containers, orquestração de containers, serviços de descoberta de serviços, API gateways, entre outros.

3. Atualmente, diversas aplicações escritas em Java estão deixando de serem desenvolvidas para rodar em servidores (JBoss, Tomcat), adotando ferramentas que disponibilizam um servidor embutido na própria ferramenta. Quais são os principais desafios ao se tomar uma decisão dessas? Justifique sua resposta.

   Essa é uma prática que tem aumentado à medida que a arquitetura orientada a microservice tem se popularizado. Pois como o micro serviço executará em um container leve como um container docker e kubernetes, não há necessidade de testar em servidores de aplicação, como um Jboss. Basta executar como um container docker que possui uma imagem com o servidor de aplicação web lev ou utilizar o servidor embutido na IDE como o Jetty. Isso acelera muito o ciclo de desenvolvimento, pois para executar um servidor embutido leve é mais rápido e consome menos recursos que o um servidor standalone completo.

4. Teste prático (em anexo)