Processo de Debugging em uma Aplicação Java Complexa

1. Primeiros Passos

- **Analisar os logs**: A primeira coisa que faço é verificar os logs da aplicação, se disponíveis. Eles frequentemente fornecem informações valiosas sobre falhas, exceções não tratadas ou erros específicos. (Existem ferramentas como o **Logback** ou **SLF4J** podem ser usadas para gerar logs detalhados, mas geralmente uso o log de erro padrão)

- **Verificar alterações recentes**: Caso o erro tenha surgido após mudanças no código, reviso os commits ou alterações no código-fonte para ver se algum comportamento inesperado foi introduzido e quem pode ajudar caso o problema se mostre complexo.

- **Testar a hipótese**: A primeira verificação consiste em entender o escopo do problema. Caso seja possível, tento reproduzir o erro localmente, verificando as entradas de dados e as condições que possam ter levado à falha.

2. Ferramentas de Debugging

O uso de ferramentas adequadas é essencial para um diagnóstico eficiente. Aqui estão as principais ferramentas que costumo utilizar:

- **IDE (IntelliJ IDEA)**: Oferece depurador robusto integrado. Permitem o uso de **breakpoints**, inspeção de variáveis em tempo de execução, visualização da pilha de chamadas e análise de fluxo de execução. No IntelliJ IDEA, eu gosto muito da visualização de variáveis e da execução passo a passo, o que ajuda a isolar rapidamente a origem do erro. Uso IntelliJ IDEA no trabalho.

- **VSCode**: Apesar de não ser uma IDE robusta como o IntelliJ possui plugins que permitem o debugging. Uso VSCode em casa.

- Existe ainda outras IDE's que auxiliam de debug como: **JProfiler**, **JProfiler**, **VisualVM** e **JDBC Profiler** para inspecionar as consultas SQL e verificar se há lentidão na comunicação com o banco de dados.

3. Técnicas de Debugging

As técnicas de debugging variam conforme o tipo de problema, e a escolha depende do contexto:

- **Breakpointing e Execução Passo a Passo**: Para problemas de lógica, como loops infinitos ou exceções inesperadas, uso **breakpoints** nas áreas do código onde o problema parece ocorrer. Isso me permite inspecionar variáveis, entender os valores intermediários e perceber o comportamento da aplicação em tempo real.

- **Logs detalhados**: Sempre que um bug não é facilmente reproduzido ou ocorre em ambiente de produção, recorro ao uso de **logs**. Coloco logs em pontos estratégicos do código para verificar o fluxo da execução, como entradas e saídas de funções críticas. Por exemplo, adicionar logs antes e depois de consultas a bancos de dados ajuda a entender onde o erro está ocorrendo (Em conjunto com conhecimentos básicos de Docker e comandos CLI).

- **Inspeção da Pilha de Chamadas**: Quando ocorrem exceções, inspeciono a **pilha de chamadas** para entender a sequência de chamadas que levou à falha. Isso ajuda a identificar funções problemáticas e pontos onde exceções podem não estar sendo tratadas corretamente.

- Existe ainda o **Profiling** quando o problema está relacionado ao desempenho (por exemplo, uso excessivo de CPU ou memória), com ferramentas como **JProfiler** para monitorar o uso de recursos e identificar gargalos. Isso inclui monitoramento do tempo de execução de métodos e análise de alocações de memória.

4. Gerenciamento de Erros e Exceções (Pré debugging)

A forma como gerenciamos erros e exceções no código é crucial para o processo de debugging:

- **Tratamento adequado de exceções**: Em Java, é fundamental capturar exceções de maneira inteligente, utilizando blocos `try-catch` de forma adequada. Utilizo exceções personalizadas para transmitir informações detalhadas sobre o erro ocorrido, o que facilita a análise.

- **Boas práticas**: Seguir o princípio de **não suprimir exceções**. Quando uma exceção é capturada, é importante logá-la e, se possível, tratar de forma inteligente, sem esconder o problema. Além disso, faço uso de mecanismos de **fallback** (por exemplo, quando a rede está instável, pode-se tentar uma reconexão).

- **Testes automatizados**: Implemento testes unitários e de integração para detectar comportamentos inesperados antes de lançar atualizações. Os testes ajudam a pegar erros antes que o código vá para produção.

5. Monitoramento e Performance

Para otimizar a performance em uma aplicação Java, sigo as seguintes estratégias:

- **Uso de Garbage Collection**: Com o **VisualVM** ou **JProfiler**, temos o **garbage collection** para identificar vazamentos de memória e otimizar a coleta de lixo. Analise do uso de memória da JVM para garantir que não haja desperdício de recursos.

- **Monitoramento de Threads**: Problemas de **deadlock** ou threads bloqueadas podem ser difíceis de identificar, mas as ferramentas de profiling podem identificar essas condições, mostrando quais threads estão sendo consumidas em operações dispendiosas.

- **Ajustes de configuração de JVM**: **Tuning** de parâmetros da JVM (como o tamanho do heap, parâmetros de GC) pode ser necessário em aplicativos de grande porte. Ajustar esses parâmetros pode melhorar a performance em operações críticas.

6. Documentação e Colaboração

Durante o processo de debugging, **documentar as descobertas** é crucial principalmente se o problema tiver uma natureza intrinsica ou recorrente à regra de negocio ou as tecnologias utilizadas:

- **Colaboração**: Em equipe, o uso de ferramentas de gerenciamento de tarefas como **Azure DEVOPS** facilita o compartilhamento de problemas e soluções. Coloco links para branchs, outros cards, commits e detalhes sobre o erro para facilitar o diagnóstico coletivo.

7. Melhores Práticas para um Debugging Eficiente

Por fim, algumas práticas que sigo para garantir um processo de debugging eficiente são:

- **Reproduzir o erro em ambiente de desenvolvimento** antes de tentar corrigir em produção.
- **Isolar o problema**: Tentar isolar o código ou a área específica onde o erro está ocorrendo. Muitas vezes, isso envolve comentar partes do código, simular diferentes condições de entrada ou breakpoints.
- **Evitar tentativas de correção apressadas**: Apressar correções pode gerar novos problemas, onde mostra-se a importancia de testes automatizados.
- **Revisar o código de forma colaborativa**: Peço uma revisão do código para outro membro da equipe pois pode ajudar a identificar erros que eu mesmo não vi. No processo de desenvolvimento que utilizo a review por outro dev é mandatória.