Lista de Restrições

Restrição	Razão (lógica)
O software deve ser desenvolvido em Java.	Popularidade no mercado, robustez, e suporte a um amplo ecossistema de bibliotecas e frameworks, além de ser amplamente utilizado em aplicações corporativas e suporte à diversas plataformas.
O framework de desenvolvimento Java Spring Boot deve ser utilizado.	A facilidade de configuração inicial e tempo de desenvolvimento reduzido com configurações pré-estabelecidas tornam este framework o mais eficaz para esse contexto.
A biblioteca Java Swing deve ser utilizada para desenvolver a interface gráfica do sistema.	Rápida e fácil de implementar, é parte do Java Development Kit (JDK) e não requer dependências externas.
O sistema deve utilizar Oracle DataBase como o banco de dados relacional.	Altamente escalável, seguro e com alta disponibilidade. Proporciona um ambiente robusto e muito consistente, sendo capaz de efetuar transações complexas e continuar disponível mesmo em situações de alta demanda.
Deve ser compatível com servidores que possuam Java Runtime Environment (JRE) versão 8 ou superior.	Para garantir a compatibilidade do sistema com versões amplamente adotadas do JRE, evitando problemas de execução em ambientes de produção comuns.
As operações CRUD devem ser realizadas com um tempo de resposta máximo de 2 segundos.	A fim de manter uma experiência de usuário aceitável e responsiva mesmo que em situações de carga moderada.
O sistema deve implementar autenticação e autorização de usuários usando o framework Spring Security.	Para garantir que apenas usuários autenticados possam acessar ou modificar os dados, protegendo o sistema contra acessos não autorizados e atendendo aos requisitos básicos de segurança.
O software deve ser compatível com sistemas Windows, Linux e MacOS.	Java é uma linguagem multiplataforma, portanto o sistema deve ser capaz de rodar

	em diferentes ambientes operacionais, garantindo flexibilidade no uso e no deploy.
Todas as bibliotecas e frameworks utilizados devem ter licenciamento compatível com uso comercial (exemplo: MIT, Apache).	Para garantir que todas as dependências do sistema estejam em conformidade com as políticas de licenciamento.
O sistema deve ser capaz de suportar um crescimento de até 10.000 registros por entidade sem perda significativa de performance.	Para assegurar que o sistema possa escalar conforme o crescimento dos dados sem perder a usabilidade.
A interface do usuário deve ser simples e intuitiva.	Facilitar o uso para usuários finais, garantindo que mesmo usuários com pouca ou nenhuma experiência técnica sejam capazes de operar o sistema com facilidade.
O sistema deve estar em conformidade com a Lei Geral de Proteção de Dados (LGPD) - a Lei federal n. 13.853 de 2019, de 8 de julho.	Garantir que o sistema atenda às exigências legais relacionadas ao tratamento de dados pessoais, evitando penalidades legais e assegurando a privacidade dos usuários.
O desenvolvimento deve seguir (quando possível) práticas de Clean Code e padrões de projeto onde aplicável.	Assegurar que o código seja limpo, legível e fácil de manter, além de permitir futuras expansões e melhorias.
Todas as funcionalidades devem ter cobertura de testes unitários e de integração.	Garantir a confiabilidade do sistema, minimizando o risco de bugs e problemas em produção, e facilitando a manutenção do sistema a longo prazo.
O projeto deve seguir o modelo de ramificação Git Flow.	Modelo de ramificação popular que organiza o desenvolvimento em branches como *main*, *develop*, *feature*, *release*, e *hotfix*, facilitando o controle de versões e a integração contínua. Ele garante que o código em produção (branch *main*) seja sempre estável e que novas funcionalidades sejam desenvolvidas e testadas de forma isolada antes da integração.
Os nomes das branches devem seguir um padrão específico, como feature/, bugfix/, release/, etc.	Identificação rápida do propósito de cada branch, melhorando a comunicação entre os

	desenvolvedores e a organização do repositório.
As mensagens de commit devem ser claras e seguir o padrão Conventional Commits.	Mensagens de commit padronizadas ajudam na rastreabilidade das mudanças, facilitando a revisão de código e o entendimento do histórico do projeto.
Todo código deve passar por revisão via Pull Request (PR) antes de receber um merge para a *main*.	Revisões de código garantem a qualidade do código, ajudam a identificar bugs e mantêm a consistência com as práticas de desenvolvimento definidas.
O projeto deve ser configurado com integração contínua (CI) utilizando o GitHub Actions para rodar testes automatizados a a cada commit.	Garante que o código não quebre funcionalidades existentes e que a qualidade seja mantida ao longo do desenvolvimento.
O repositório GitHub deve ser versionado seguindo o Semantic Versioning (SemVer).	Define uma convenção para atribuir números de versão (ex: 1.0.0), onde mudanças de API incompatíveis, funcionalidades adicionadas e correções de bugs são claramente indicadas pela mudança de números de versão.
Toda mudança significativa no código ou arquitetura deve ser documentada em um arquivo CHANGELOG.md e na wiki do repositório seguindo o modelo Keep a Changelog.	Ajuda na comunicação com outros desenvolvedores e usuários finais, além de facilitar a manutenção do sistema a longo prazo.
O merge para a branch main deve ser realizado apenas após passar por um processo de revisão e aprovação de, pelo menos, dois desenvolvedores, e todos os testes de CI devem ser aprovados.	Garante que o código que vai para produção seja de alta qualidade e sem problemas conhecidos, evitando interrupções ou bugs em produção.
O sistema deve seguir o padrão de arquitetura MVC (Model-View-Controller).	Facilita a manutenção do código, separando as responsabilidades entre a lógica de negócio (Model), a interface do usuário (View), e o controle de fluxo (Controller). Isso melhora a organização do código e torna o sistema mais modular e expansível.

O código deve ser organizado em diretórios distintos para Model, View, e Controller.	Manter uma estrutura de diretórios clara e separada de acordo com os componentes do MVC facilita a navegação no projeto e a colaboração entre desenvolvedores, permitindo que cada camada seja modificada sem impacto direto nas outras.
A camada View não deve acessar diretamente a camada Model; toda comunicação entre a View e o Model deve ser mediada pelo Controller.	Garante que a interface do usuário (View) não tenha dependências diretas da lógica de negócio (Model), promovendo a separação de responsabilidades e permitindo que a interface e a lógica de negócio possam evoluir independentemente.
O Controller deve ser responsável por receber as entradas do usuário, processar as interações entre View e Model, e atualizar a View de acordo com os resultados.	Garantir que o Controller mantenha a lógica de aplicação e controle do fluxo de dados, mantendo a camada Model focada na manipulação de dados e a View na apresentação da interface do usuário.
O Model deve ser responsável por todas as interações com o banco de dados, encapsulando a lógica de persistência.	Centralizar a lógica de acesso a dados no Model garante que todas as operações de banco de dados sejam consistentes e facilita a implementação de alterações na persistência de dados sem impactar outras camadas do sistema.
Cada camada do MVC deve ser testável de forma isolada, com a camada Model testada por meio de testes unitários, e as camadas Controller e View testadas por meio de testes de integração e de interface.	A testabilidade isolada de cada camada garante que bugs possam ser identificados e corrigidos em sua origem, mantendo a robustez e a qualidade do código.
A comunicação entre Model e Controller deve ser feita por meio de DAOs (Data Access Objects) para promover o encapsulamento e a reutilização de código.	Separar a lógica de acesso a dados da lógica de aplicação facilita a manutenção, promove o reuso de componentes, e melhora a escalabilidade do sistema.
O sistema deve seguir os princípios S.O.L.I.D.	Garantir que o sistema seja construído de maneira robusta, modular e fácil de manter.
Cada classe deve ter uma única responsabilidade, ou seja, deve haver apenas um motivo para que a classe seja	Promove a clareza do código e facilita a manutenção ao garantir que as classes tenham funções específicas e bem

modificada (Restrição de Responsabilidade Única).	definidas. Isso reduz o acoplamento e torna o sistema mais modular.
O código deve ser projetado de forma que as classes e funções sejam abertas para extensão, mas fechadas para modificação (Restrição Aberto/Fechado).	Isso permite que novas funcionalidades sejam adicionadas ao sistema sem alterar o código existente, minimizando o risco de introduzir bugs e facilitando a expansão do sistema.
Subclasses ou classes derivadas devem ser substituíveis por suas classes base, sem alterar o comportamento esperado do programa (Restrição de Princípio de Liskov).	Garantir que subclasses possam ser usadas em qualquer contexto em que a classe base é esperada, ajuda a manter a integridade do sistema e a evitar problemas de compatibilidade e comportamento inesperado.
Interfaces devem ser específicas e focadas, evitando que classes sejam forçadas a implementar métodos que não utilizam (Restrição de Segregação de Interface).	A segregação de interfaces promove a modularidade e a clareza, evitando que classes sejam sobrecarregadas com métodos desnecessários, o que facilita a evolução e manutenção do código.
Módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações (interfaces ou classes abstratas). As abstrações não devem depender de detalhes; os detalhes devem depender das abstrações (Restrição de Inversão de Dependência).	Facilita a substituição de componentes, promove a reutilização de código, e simplifica o teste do sistema, pois permite o uso de mocks e stubs em testes unitários.