Para detalhes técnicos para subir a aplicação leia o arquivo README.txt, que se encontra na raiz do projeto.

Explicação sobre a modelagem e funções do sistema

Ao modelar os dados do cliente procurei fazer de forma simples, e não coloquei colunas como documento e telefone, tenho conhecimento que numa aplicação real, provavelmente seria necessário, mas tecnicamente não acrescentaria mais nada e traria mais trabalho com verificações de dados e tamanho. Acrescentei o endereço para mostrar uma complexidade na modelagem, e também trazer funcionalidade úteis de integração com uma API externa, mas pelo lado da simplicidade o sistema permite cadastrar o endereço apenas com o país (que inclusive pode ser colocado pelo próprio sistema), isso também para evitar tantas verificações maiores que tecnicamente não traria mais ganhos. Entretanto sei que numa funcionalidade real, seria obrigatório a maioria dos campos.

De novidade há funcionalidades úteis para ajudar a cadastrar o endereço que pode trazer a lista de estados brasileiros, as cidades, dado o estado, e também a busca de endereço por Cep, coisas bem úteis numa aplicação real.

No endereço, optei por colocar a informação de país, pois já vi aplicações reais que precisaram internacionalizar e foi extremamente difícil por não ter a simples informação de país, coisa que pode ser muito útil na ciência de dados também, mesmo que inicialmente se prever apenas o uso no Brasil. A informação de cep ou zipcode, é uma String, mas não permite traços e pontos do formato no cadastro, só os 8 dígitos (entretanto pode mandar o cep formatado que sistema limpa). Optei por uma String ( e varchar no banco) pois há ceps que começam com zero, e os tipos numéricos cortam o zero inicial (já vi muitos sistemas com problemas por não prever essa particularidade, a mesma coisa com o cpf, também há os que começam com zero, o meu por exemplo).

Já o número do endereço optei por uma modelagem diferente e coloquei o tipo numérico, pois como há o campo complemento aqueles endereços tipo 101A, seriam ser contemplados no complemento. Também não modelei o tipo do logradouro, pois essa informação pode ser colocada junto com o logradouro tipo avenida Brasil ou rua Santos Dumont.

No cliente coloquei um campo status, que a princípio pode ser usado para habilitar e desabilitar os clientes, fazendo exclusão lógica, poderia ter usado boolean para isso mas optei por um enum que podem ser estendido, pois já vi projetos em que começam colocando um boolean para isso, aí veem a necessidade da ação cancelar, ai criam outro boolean, cancelado, depois, veem a necessidade de uma nova ação para inadimplente por exemplo aí colocam um novo boolean, uma nova coluna, no fim, há n booleans e uma lógica enorme para saber o estado do cliente, sendo que se tivesse sido mapeado como um enum desde o início, teria menos colunas, uma lógica bem mais tranquila e um código mais limpo.

Optei por usar o Spring Data JPA para fazer a paginação e as consultas de clientes, a vantagem é a facilidade na paginação. A desvantagem é que numa aplicação real as telas de filtros com o usuário escolhendo filtros e com alguns campos não obrigatórios, sendo que o Spring Data dificulta esses filtros mais “dinâmicos”, tendo que criar uma Specification ou usar uma estratégia diferente como a Entity Manager do Java. Entretanto, como iria trazer uma complicação grande ao código, além de não fazer parte especificamente dos requisitos optei por manter o Spring Data simples, assim é possível a consulta do cliente por e-mail ou nome, ou nome e e-mail ou por data de nascimento, que já traz uma boa gama boa de consultas, apesar de um tanto fixas.

Não vejo como boa prática retornar um objeto Entity pelo Controller, assim todos os objetos são convertidos para um Vo ( mesmo que tenha os mesmos campos) antes de retornado. A maioria das queries de consulta, optei por escrever através da @query em vez de só ser chamada pelo nome do método automaticamente ( recurso permitido pelo Spring Data), pois pela minha experiência, essas queries “automáticas” fazem queries muitas vezes mais pesadas e repetidas do que o necessário e sobrecarregam o sistema. Já vi sistema cair por causa de uma query “simples” de cliente inclusive.

Usando Junit criei testes para o sistema. Mas em vez de usar mocks, preferi fazer testes mais abrangentes, que testam a nível de service, chegando inclusive no repository, e acessando o banco de dados, acredito que os testes assim são mais efetivos, e realmente testam o fluxo dos objetos.

Não usei lombok porque não gosto da legibilidade de projetos que o usam. E também não usei o liquid base, porque já vi projetos que na hora de subir ficam travados por causa dele ( dá lock no banco).