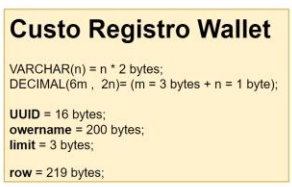
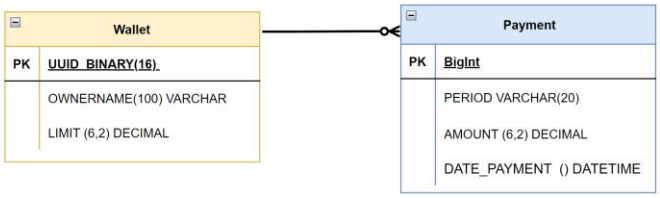
Stack do projeto, na ordem: *MySQL, Flyway, Kotlin, Ecossistema Spring, Docker e Github actions*

Modelo de dados da aplicação e aproximadamente o custo de cada linha:

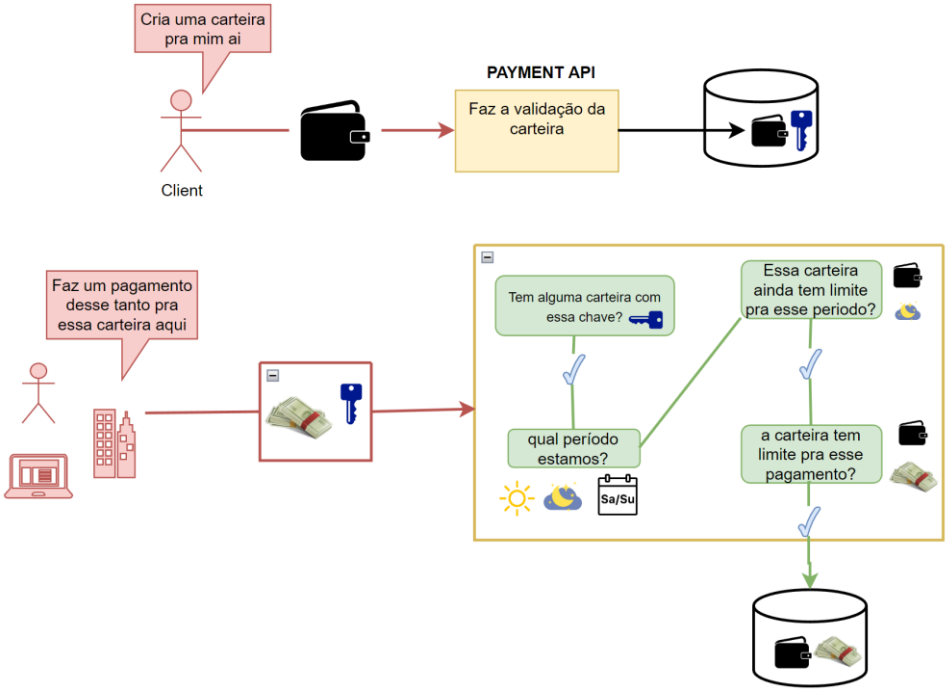




É muito importante enfatizar que as linhas estão com valores deduzidos, não inclui cálculo de índices e muitas outras variáveis. Mas o ponto principal aqui é enfatizar o quão "pequeno" é o banco de dados, e, portanto, faz sentido usar uma abordagem relacional (usei o MySQL por afinidade, mas qualquer outro banco relacional poderia servir).

O propósito da API, de maneira simplificada:

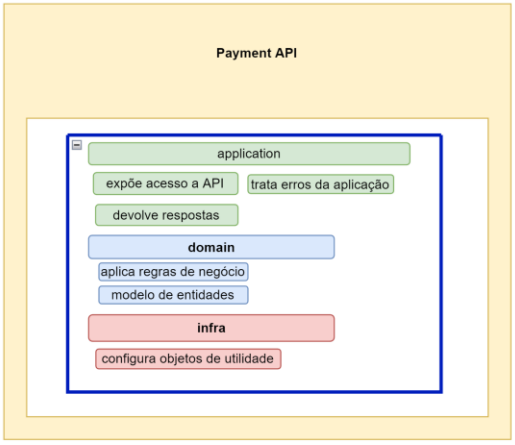
A responsabilidade da API é gerenciar todas as regras que estão envolvidas em pagamentos realizados para uma carteira:

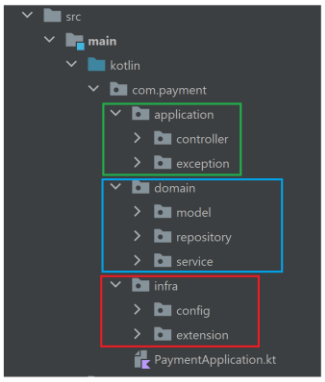


Arquitetura Geral:

A organização primária do projeto são os pacotes. Temos um total de 3 “superpacotes” (os principais), eles visam organizar o sistema de acordo com algumas responsabilidades. São eles:

- **Pacote de “application” :** nesse paco é onde você encontra as classes que dizem respeito a responsabilidade da aplicação, note que não estou me referindo ao DOMÍNIO (esse tem mais detalhes de implementação). Por exemplo, é obrigação da minha aplicação tratar as exceptions, portanto, meu pacote de application tem o paco que contém as exceções.
- **Pacote de “domain”:** esse merece uma descrição mais detalhada, e na próxima seção desse documento ele é melhor descrito. Por hora basta saber que é aqui que você vai encontrar regras de negócio, modelos de negócio e repositórios.
- **Pacote de “infra”:** A ideia desse pacote é conter configurações de infraestrutura do projeto que não são obrigação da aplicação. Por exemplo, a classe de configuração swagger.

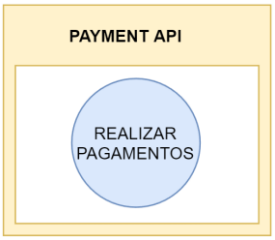




Alguns conceitos de DDD usados:

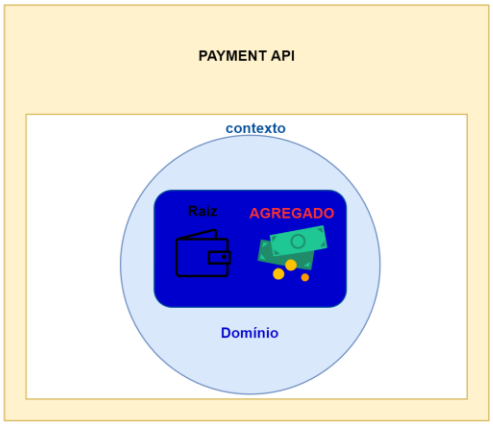
contexto, domínio, entidades (root e agregado) e como isso se reflete em código:

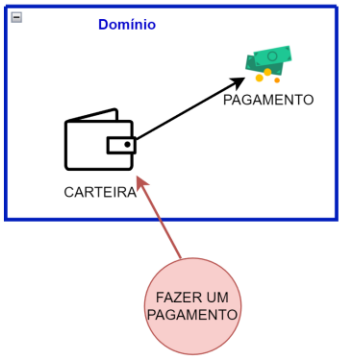
Sabendo como a aplicação deveria funcionar, já dá pra começar a aplicar o básico de DDD, a primeira coisa é definir um contexto. Olhando o diagrama fica bem claro que o único contexto da aplicação é ***fazer um pagamento:***



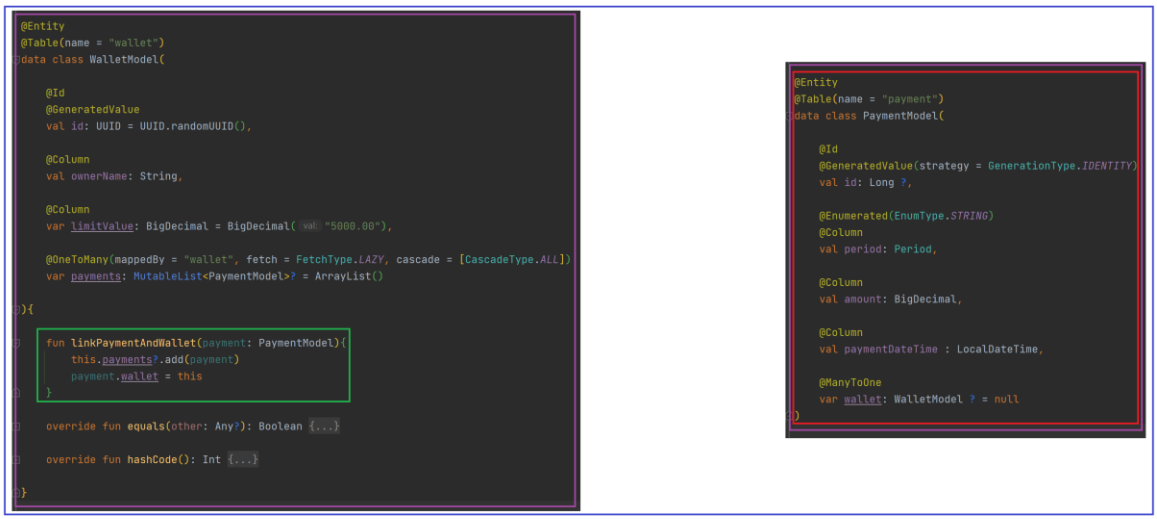
Um **pagamento** é feito único e exclusivamente para uma **carteira**. Ok, isso me gera ***duas entidades*** que serão MODELOS para representar o **domínio de pagamento**, mas tem um ponto aqui, um **pagamento NÃO EXISTE** sem uma **CARTEIRA**. Isso quer dizer que meu **PAGAMENTO é um AGREGADO** da minha **CARTEIRA(raiz).**

Aqui quando eu digo que um modelo é agregado do outro, e que esse outro é uma raiz, eu estou querendo dizer que **toda modificação/acesso nos modelos do domínio deverão ser realizados através da raiz:**





Em código, esses **MODELOS me geraram duas entidades.** Repare **que a responsabilidade de vincular um pagamento a uma carteira, e uma carteira a um pagamento é TOTALMENTE DO MEU MODELO RAIZ**:



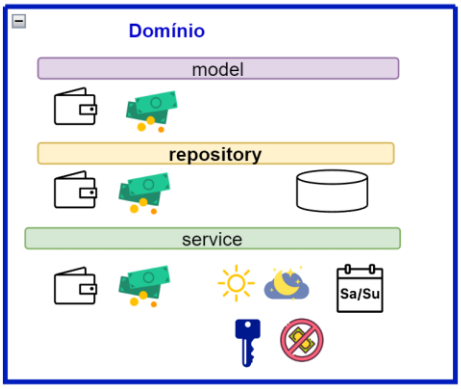
```kotlin
@Entity
@Table(name = "wallet")
data class WalletModel(

    @Id
    @GeneratedValue
    val id: UUID = UUID.randomUUID(),

    @Column
    val ownerName: String,

    @Column
    var limitValue: BigDecimal = BigDecimal( val: "5000.00"),

    @OneToMany(mappedBy = "wallet", fetch = FetchType.LAZY, cascade = [CascadeType.ALL])
    var payments: MutableList<PaymentModel>? = ArrayList()

){

    fun linkPaymentAndWallet(payment: PaymentModel){
        this.payments?.add(payment)
        payment.wallet = this
    }

    override fun equals(other: Any?): Boolean {...}

    override fun hashCode(): Int {...}

}
```

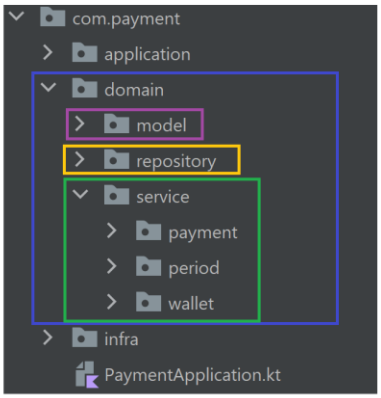
```kotlin
@Entity
@Table(name = "payment")
data class PaymentModel(

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long ?,

    @Enumerated(EnumType.STRING)
    @Column
    val period: Period,

    @Column
    val amount: BigDecimal,

    @Column
    val paymentDateTime : LocalDateTime,

    @ManyToOne
    var wallet: WalletModel ? = null
)
```

repositórios e serviços e como isso reflete na estrutura do projeto:

Outro conceito aplicado do DDD aqui foi a relação dos repositórios, que basicamente descrevem comportamentos relacionados a persistência/leitura/atualização das entidades. Os serviços são a lógica por trás de determinada ação, então minha ação de fazer um pagamento é um serviço.

Usando a **estrutura de pacotes** eu isolo o **domínio** da minha aplicação, meu **pacote de domínio** vai conter os **modelos** que representam o ***domínio de fazer um pagamento,*** os ***serviços que representam a lógica*** de fazer um pagamento e os ***repositórios que descrevem ações de CRUD*** do domínio de fazer um pagamento:


Domínio
model
repository
service
Sa/Su


com.payment
application
domain
model
repository
service
payment
period
wallet
infra
PaymentApplication.kt