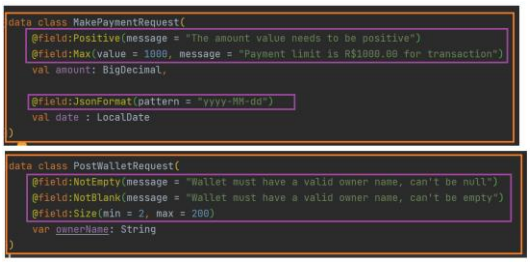
Validação de entrada das requisições

Todos os pontos de entrada da aplicação são **protegidos de requests que não seguem as regras**. **Carteiras não podem ser criadas se não tiverem um nome adequado**, pagamentos **não podem ser feitos com valores abaixo de 1 ou data com formato inválido.**

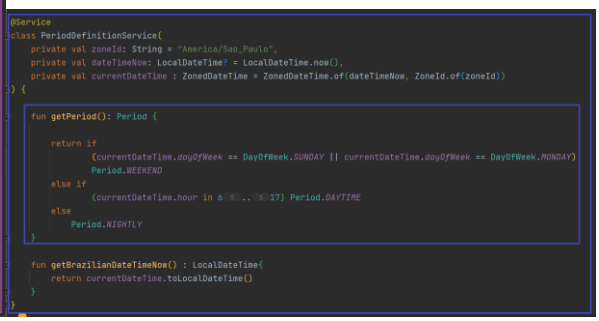
Todas essas verificações de entrada foram feitas usando o ***Spring Validation***, um projeto do spring específico para esse tipo de ação. As **classes de request** usam as **anotações que definem como o campo deve ser validado**, anotando ***os métodos do controller com @Valid eu digo para a validação ser feita*** com base nos campos anotados:



Como a atribuição dos períodos se comporta

Os períodos definidos são representados por um **ENUM**, esse ENUM além de conter os períodos funciona como uma *fábrica para a estratégia de pagamentos (é o próximo tópico)*. A decisão de tratar o fim de semana como um período foi para facilitar a centralização das estratégias de pagamento.

A ideia em atribuir o período funciona utilizando o **horário de Brasília (UTC-3),** assim que a requisição de pagamento entra no sistema, a **classe de serviço de definição de período** entra em ação, ela sempre é instanciada com a ***hora local no momento***, que ela usa como base **para definir o período:**



O período **do pagamento é então atribuído**, dessa maneira o pagamento pode ser processado corretamente. Outra coisa importante aqui é **ter uma data e hora no pagamento**, isso permite flexibilidade caso seja necessário saber o horário de um pagamento no período de fim de semana:

```kotlin
@PostMapping("/{walletId}/payments")
@ResponseStatus(HttpStatus.OK)
fun makePayment(@RequestBody @Valid request:MakePaymentRequest, @PathVariable walletId: UUID){
    val wallet = walletService.findById(walletId);
    val brazilTime = periodDefinitionService.getBrazilianDateTimeNow().toLocalTime()
    val payment = request.toPaymentModel(periodDefinitionService.getPeriod(), brazilTime)
    paymentService.makePayment(wallet, payment)
}
```

```kotlin
fun MakePaymentRequest.toPaymentModel(period: Period, brazilTime: LocalTime): PaymentModel{

    return PaymentModel(
        id = null,
        amount = this.amount,
        period = period,
        paymentDateTime = LocalDateTime.of(this.date, brazilTime),
        wallet = null,
    )
}
```

Como funciona a estratégia de pagamento

Basicamente o pagamento funciona validando se os limites foram atingidos, pra isso existe **uma query que retorna o total de pagamentos feitos a uma carteira num determinado dia e período**. Com essa informação eu **deixo minha estratégia** definir se o pagamento pode ou não ser feito:

```kotlin
@Service
class MakePaymentService(
    private val crudPaymentService: CrudPaymentService,
    private val crudWalletService: CrudWalletService,

) {

    @Transactional
    fun makePayment(wallet: WalletModel, payment: PaymentModel) {

        val periodPaymentStrategy : PaymentStrategy = payment.period.getPeriodStrategy()
        val today = payment.paymentDateTime.toLocalDate()
        val amountUntilNow = crudPaymentService.getTotalPaymentsByPeriod(wallet.id, today, payment.period)
```

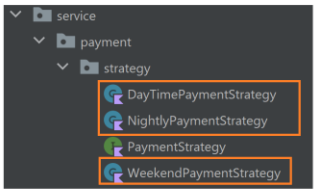
Eu me baseei no padrão de ***strategy*** para trabalhar com as regras de pagamento, onde as **implementações concretas** se baseiam em períodos. A ***estratégia funciona simplesmente dizendo se um pagamento pode ou não ser realizado em um determinado período*** (se já atingiu o limite). Esses limites estão presentes em outro **ENUM**:

```kotlin
interface PaymentStrategy {

    fun getCurrentLimit () : BigDecimal

    fun walletStillHavePeriodLimit(totalPaymentsAlreadyMade: BigDecimal) : Boolean

}
```




```kotlin
enum class PeriodLimitValue {

    DAYTIME_LIMIT {
        override fun getLimit(): BigDecimal = BigDecimal( val: "4000.00")
    },
    NIGHTLY_LIMIT{
        override fun getLimit(): BigDecimal = BigDecimal( val: "1000.00")
    },
    WEEKEND_LIMIT {
        override fun getLimit(): BigDecimal = BigDecimal( val: "1000.00")
    };

    abstract fun getLimit() : BigDecimal
}
```

Depois disso ocorre **outra verificação, essa pra saber se a carteira ainda tem limite pra receber o montante do pagamento**. Se tudo ocorrer bem, o pagamento é **descontado do limite da carteira e as informações são persistidas:**

```
if(!periodPaymentStrategy.walletStillHavePeriodLimit(amountUntilNow)) throw PeriodLimitReachedException(
    Errors.TP203.message.format(payment.period, periodPaymentStrategy.getCurrentLimit()),
    Errors.TP203.code
)

if(!walletHasLimit(wallet.limitValue, payment.amount)) throw WalletLimitReachedException(
    Errors.TP204.message.format(wallet.limitValue),
    Errors.TP204.code
)

wallet.limitValue = newLimitAfterDiscount(wallet.limitValue, payment.amount)
wallet.linkPaymentAndWallet(payment)

crudWalletService.save(wallet)
```

Garantindo que as carteiras tenham limites diários

Para garantir que as carteiras tenham seus limites diários resetados eu usei o **scheduled do Spring** para **agendar um reset do limite** das carteiras sempre as **00:0:**

```
@Service
class ResetWalletLimitService(
    private val FIRST_LIMIT_OF_THE_DAY: BigDecimal = BigDecimal( val: "5.000"),
    private val crudWalletService: CrudWalletService
) {

    companion object{
        private const val  CRON_DAILY_RESET : String = "0 0 0 1/1 * ? *"
    }

    @Scheduled(cron = CRON_DAILY_RESET)
    fun resetWalletsLimits(){
        crudWalletService.resetWalletsLimits(FIRST_LIMIT_OF_THE_DAY)
    }
}
```

Aqui eu fiquei meio frustrado, pois o ideal na minha concepção seria ter uma rotina batch que salvasse o estado das carteiras antes do reset, mas eu não tive tempo de estudar a solução e implementar (***queria estudar o Spring Batch)***, então fiz uma consulta muito simples que pode provavelmente não ser performática.]

Uma cobertura completa de código

No escopo de testes da aplicação eu utilizei os **testes unitários na camada de serviço** e **testes de integração referente a querys que eu escrevi e todo o fluxo dos controllers**. A aplicação tem uma cobertura de todo o funcionamento, de ponta a ponta.

Você pode executar o comando ***"./gradlew check jacocoTestCoverageVerification jacocoTestReport"*** para gerar o seguinte relatório de cobertura no caminho **build/reports/jacoco/test/html/index.html. MAS**

LEMBRE-SE, TESTES DE INTEGRAÇÃO USAM O BANCO DE DADOS, ENTÃO CERTIFIQUE-SE DE LEVANTAR O CONTAINER DO BANCO DE DADOS: