**Introdução**

O início do trabalho foi pensado em garantir também os pontos extras e, para isso, precisou-se da escolha de uma web framework C++. Foram pensados em dois deles, o CWF (C++ Web Framework) e o Crow Web framework, sendo o último escolhido por parecer ser mais simples. O banco de dados escolhido foi o MySQL.

Para unir todos esses programas e fazer de uma forma que fosse possível uma rápida edição entre os integrantes do grupo, foi usado o docker-compose em união com o GitHub; assim, criava-se um container Docker com as seguintes aplicações:

* Theia (IDE online)
* Crow (WebFramework c++ com todas suas bibliotecas integradas)
* Nginx (responsável pelo frontend e o proxy reverso (para comunicação com o crow))
* MySQL (Para o banco de dados)
* PHPMyAdmin (Para gerenciamento do banco de dados)

Desenvolvimento

1. Docker-Compose/Interface

Para a interface Web, optou-se utilizar a “game engine” Construct 3, uma vez que ela atendia todos os requisitos para o trabalho. Como o foco do trabalho em si não era a programação do frontend, foi utilizada essa ferramenta justamente por não requirir muito aprendizado e ser simples.

Ao iniciar a instalação e integração desses containers, enfrentou-se grandes dificuldades para o aprendizado do framework e para a integração do docker, o que tomou muito tempo dos integrantes.

1. C++

Como foi utilizada uma interface Web, não se precisou tratar erros, uma vez que ela somente permitia enviar dados e modificá-los com eles já sendo tratados.

Como esse trabalho foi uma extensão do TP3, utilizou-se todas as classes criadas e foram implementadas novas classes com o intuito apenas de acrescentar/modificar classes já existentes. O uso de herança e polimorfismo foi essencial para essa parte e as classes que utilizaram esses mecanismos foram as classes:

* 1. ContaTP3 (Conta)

Invés de criar 2 classes diferentes para Conta Corrente ou poupança acreditou-se ser melhor separa-las apenas através de um atributo protegido que definia 1 para conta corrente e 13 para poupança.

Foi incluído também um atributo protegido que definia o limiteCredito, assim quando a conta era do tipo Conta Corrente, existia um limite que podia ser definido pelo Usuário, caso fosse do tipo Poupança, esse limite era 0.

Assim, foi preciso usar polimorfismo apenas na função DebitarValor, uma vez que a mesma agora precisava do limiteCredito para seu funcionamento e com isso, o resto das funções necessárias foram herdadas da classe Conta

* 1. ClienteTP3(Cliente)

A Classe cliente precisava apenas de um ID para o cliente, assim foi feita a herança de todas as funções da classe Cliente e no construtor da classe ClienteTP3 foi adicionado esse recurso, adicionando também a função getClienteID para retornar qual era o ID do cliente.

* 1. BancoTP3(Banco)

Nessa Classe precisou-se principalmente modificar os vetores do tipo Cliente e Conta para suas versões modificadas ClienteTP3 e ContaTP3 respectivamente e modificar também todos os usos de funções que usava classes antigas e adapta-las para as novas Classes.

1. Integração Crow

Essa foi a parte mais importante do trabalho, ela fazia a integração frontend com backend, utilizando o WebFramework Crow e ainda tratava os erros (junto com o construct 3), caso os dados fossem inválidos ou diferentes.

Como o construct 3 realizava o trabalho das requisições HTTP GET, não era necessário que o cliente digitasse os comandos no Browser, os exemplos a seguir mostram os endereços só para tornar o exemplo mais claro.

Um exemplo de uma função que ao digitar no Browser localhost/api/listarClientes, retornava em JSON (o que tornava mais fácil o tratamento de dados no construct 3).



Outro exemplo que já tratava alguns erros é o exemplo da Criar Conta. Para a criação da conta ele precisaria do IDBanco, IDCliente, TipoConta e LimiteCredito, e caso ele não tivesse alguns desses parâmetros essenciais (como idCliente e TipoConta) ele não executava a função (que caso fosse executada sem esses parâmetros pararia o programa) e retornava uma string com o erro apresentado.

