**Ponto Calculator**

**Data: 19/12/2023**

# ATIVIDADE 1 (10% da Nota Final)

## Contextualização:

Nosso sistema trata-se de um aplicativo mobile de registro de ponto eletrônico, desenvolvido inicialmente para a plataforma Android. O aplicativo permite que os usuários cadastrem-se no aplicativo e a partir desse momento já é possível realizar o login e registrar seus pontos quando necessário (entrada e saída da jornada de trabalho), que será registrado no servidor em que roda o sistema back-end (simulando um servidor do RH de uma empresa), que deve ser acessível mesmo se o funcionário estiver fora da empresa, em caso de viagens institucionais por exemplo.

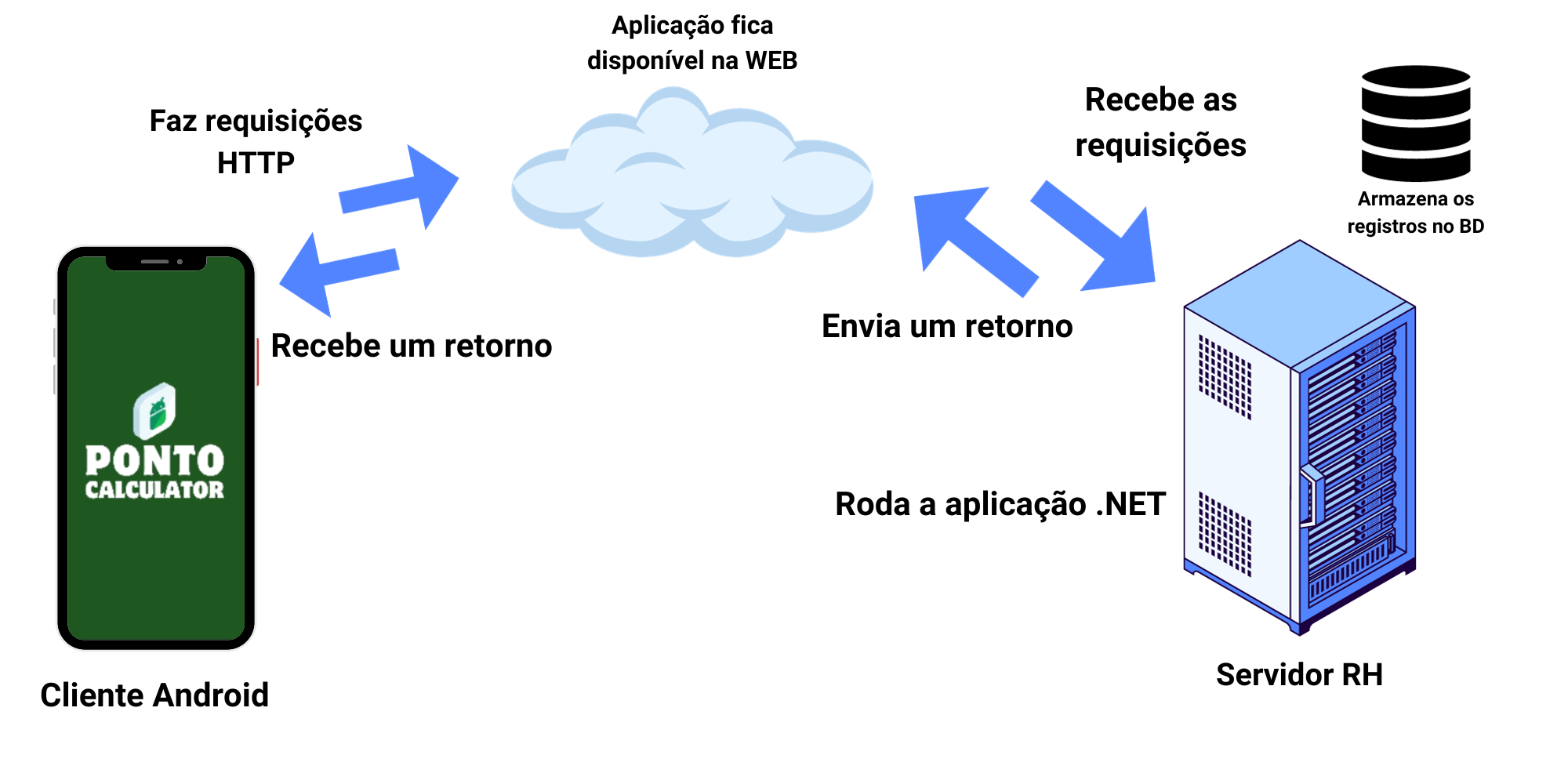
Sempre visando a segurança do usuário, esses sistema é protegido por um sistema desenvolvido em C# (.NET) que além de possibilitar o funcionamento do APP, possui autenticação via Token JWT, criptografado e com tempo para expirar. Assim, o usuário está protegido de que alguém possua seus dados em caso de furto do aparelho.

Funcionamento do aplicativo:

* O usuário inicia o aplicativo em seu dispositivo Android;
* No aplicativo, o usuário é apresentado com uma tela de login, onde ele deve inserir suas credenciais de acesso, como nome de usuário e senha;
* O aplicativo, então, valida as credenciais inseridas pelo usuário para verificar se elas estão corretas;
* Se as credenciais estiverem corretas, o aplicativo registra o usuário como autenticado e concede acesso à funcionalidade de registro de ponto do aplicativo;
* Além disso, o aplicativo fornece recursos para criar uma nova conta ou realizar a recuperação de uma conta através do e-mail cadastrado;
* Quando o usuário deseja sair do aplicativo, ele pode escolher fazer logout, o que resulta no encerramento da sessão e na exclusão das informações de autenticação armazenadas.

O foco desse projeto é especificar e dimensionar toda parte de infraestrutura básica (hardware e software) necessária para que o aplicativo mobile funcione de forma adequada. Serão analisados quais os recursos necessários para que o APP rode na maioria dos dispositivos android e também para que o servidor back-end consiga atender as chamadas de requisições e armazenamento de dados.

## Diagrama geral do projeto proposto



## Divisão dos trabalhos

Atividade 1: Cleiton / João

Atividade 2: Lucas / Matheus

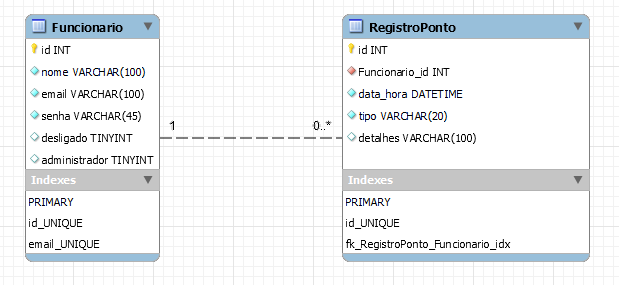
Atividade 3: Henrique

Atividade 4 e 5: Lucas, João

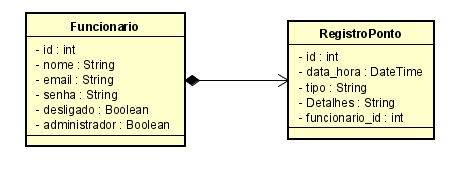
# ATIVIDADE 2 (10% da Nota Final)

## 2.1 Requisitos do sistema

O modelo Entidade Relacionamento do Ponto Calculator foi projetado com 2 tabelas, Funcionário e RegistroPonto. Dessa forma, registros de ponto podem ser criados e buscados de forma simples, seja com os dados do funcionário (id, nome, email) ou do registro de ponto (id do funcionário, data/hora, tipo).



O diagrama de classes segue a mesma lógica do modelo entidade relacionamento, sem adição de lógica de negócios que se diferencie das funções CRUD (create, read, update, delete).



O sistema foi pensado com uma aplicação backend .Net Core 7, escrita em C#, quanto do banco de dados SQL server, que tem forte compatibilidade com o framework utilizado.

Esse mesmo sistema backend pode ser facilmente containerizado utilizando o Docker, possibilitando rodá-lo em diferentes situações como em provedores de servidor cloud. Assim, o servidor da aplicação tem certa versatilidade quanto à forma em que será rodado.

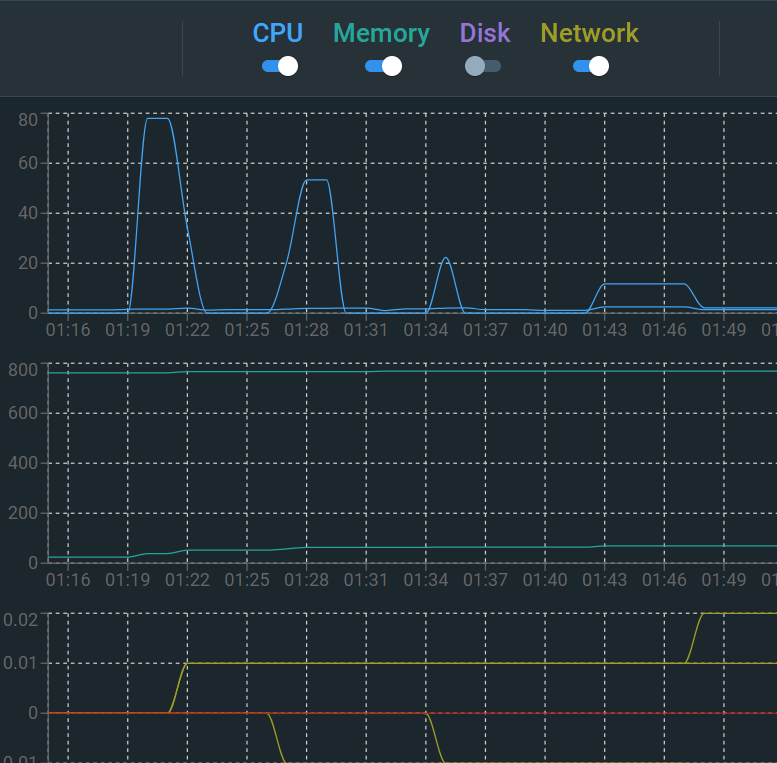
Para nossa proposta, não foi pensada nenhuma solução mais complexa de proxy, load-balancer, cache ou similar, visto que essa aplicação será voltada a pequenas empresas.

O presente projeto prevê um cliente mobile no sistema operacional Android. Porém, o backend é agnóstico sobre o cliente, podendo se comunicar com qualquer aplicação cliente através de web APIs.

Em questões de hardware, o dispositivo móvel requer ao menos 2GB de memória RAM e 17 MB de armazenamento para a instalação.

Enquanto para o servidor backend, a Microsoft estima como requisito mínimo para o SQL Server Docker container em versão Linux 2GB de espaço em disco e de memória RAM. Já a aplicação .Net gerou uma imagem de 242.16 MB e não foi observado consumo de memória RAM superior a 100MB em testes manuais. Assim totalizando 2242.16 GB de disco e 2.1GB de RAM como requisitos mínimos para rodar o sistema, dado um host linux (como por exemplo Ubuntu, usado na máquina de testes abaixo) com Docker instalado.

Testes manuais preliminares com a aplicação apenas desenvolvida até o registro, login e recuperação de senha de usuário resultaram nas seguintes métricas, mensuradas através da extensão Live Charts disponível no Docker Desktop:



Para esse teste, foram realizadas 4 chamadas consecutivas na web API da aplicação:

1. POST register: registra o usuário no banco de dados, com os parâmetros email, senha e nome.
2. POST login: faz o login do usuário, com parâmetros de email e senha, retornando um JWT.
3. GET user: retorna os dados de usuário com omissão do dash da senha, utilizando o JWT como parâmetro.
4. POST forgot-password: registra um token de recuperação de senha na tabela do usuário no banco de dados e envia um email de recuperação de senha.

De acordo com os testes acima, onde a linha mais acentuada representa o banco de dados SQL Server e a linha menos acentuada representa a aplicação .Net, notamos um uso de processamento decrescente, um uso de rede crescente, com ápice no envio do email, e um uso de memória RAM crescente. Não foi notado uso de disco mensurável, visto que o banco de dados ainda não está configurado para utilizar volumes em disco nessa versão de desenvolvimento.

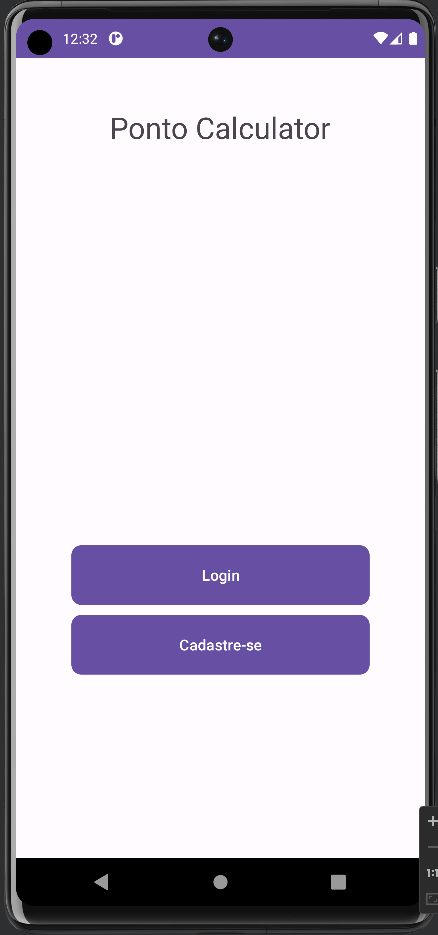
Assim, por fim, estimamos que a aplicação backend tem expectativa de não ultrapassar os requisitos de 3 GB de disco e RAM até o final de seu desenvolvimento.

# ATIVIDADE 3 (20% da Nota Final)

## 3.1 Interface do usuário

3.1.1 Tela Inicial:

* Tela de primeiro contato com o usuário, onde foram criados dois botões para interação “login” e “cadastre-se”.

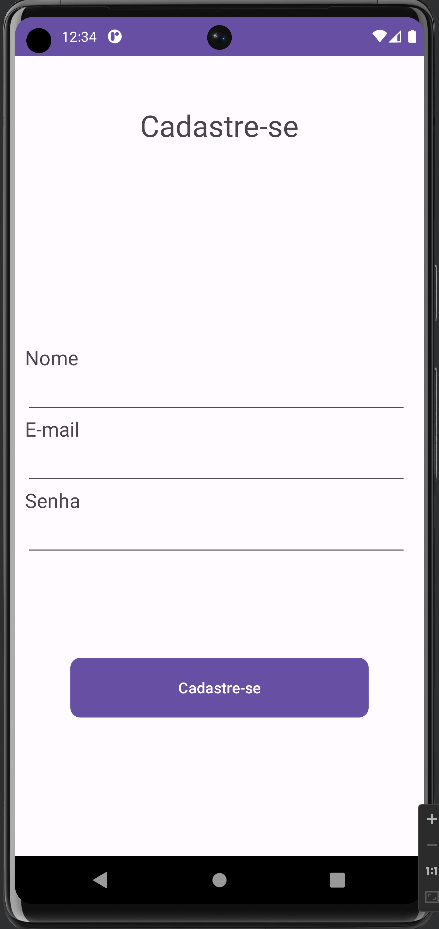


Código XML

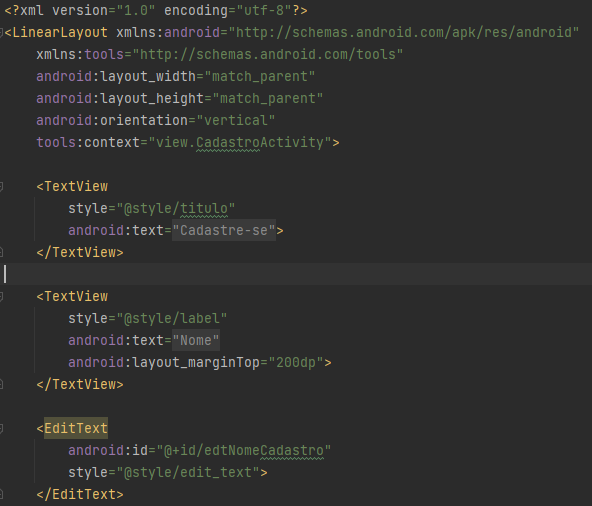


3.1.2 Tela de Cadastro

* Tela com 3 campos à ser preenchida pelo usuário, “nome” “senha” “e-mail” e com o botão “cadastrar” para seguir adiante.



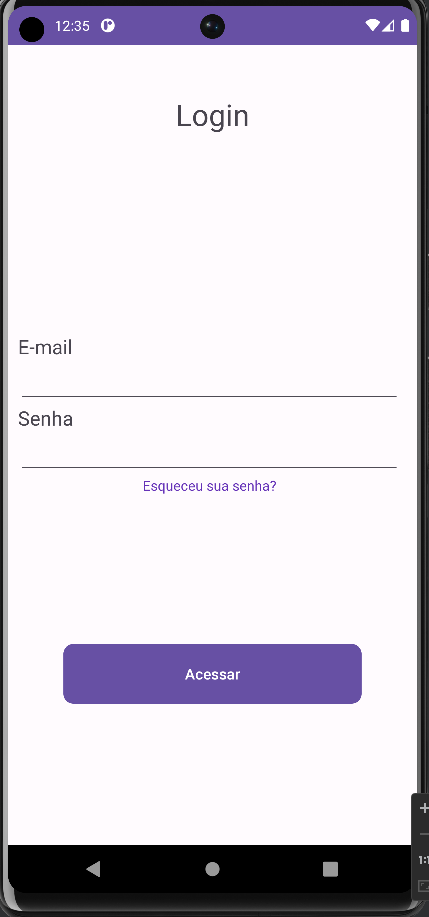
Código XML



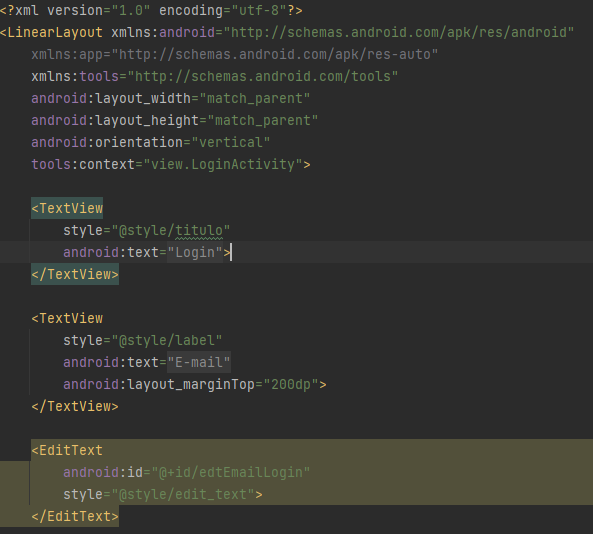


3.1.3 Tela de Login

* Tela onde o usuário já cadastrado no APP poderá acessar com seus dados gravados. (“email” e “senha”).



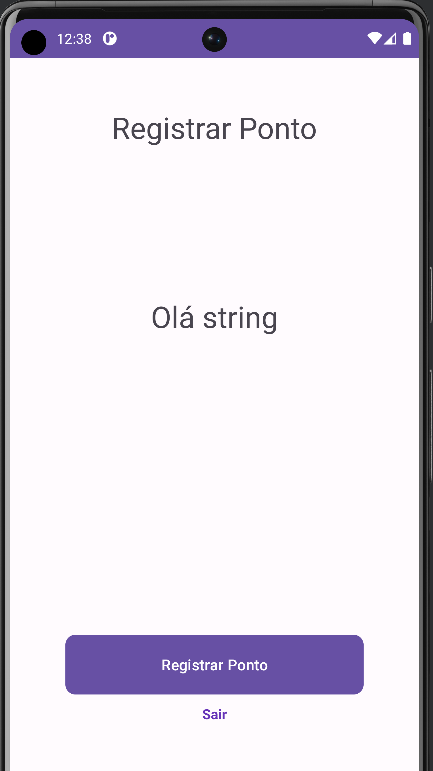
Código XML



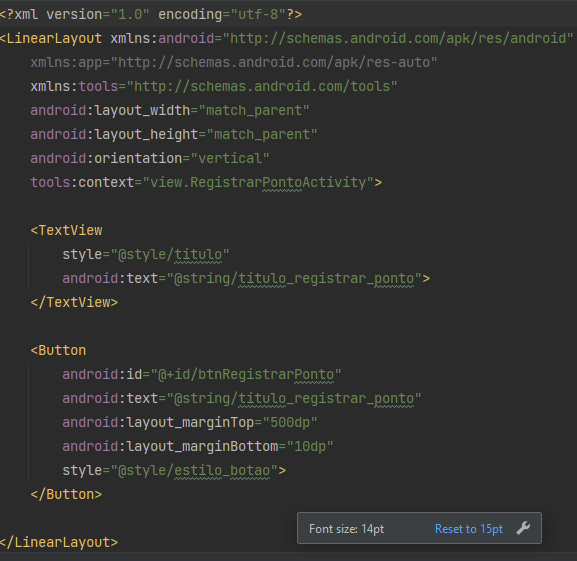


3.1.4 Tela de registro de ponto

* Tela em que o usuário fará o registro do seu ponto, nela estará escrito um pequeno texto como na tela abaixo: Olá “nome do usuário” acompanhado de um botão para efetuar a ação de registro.

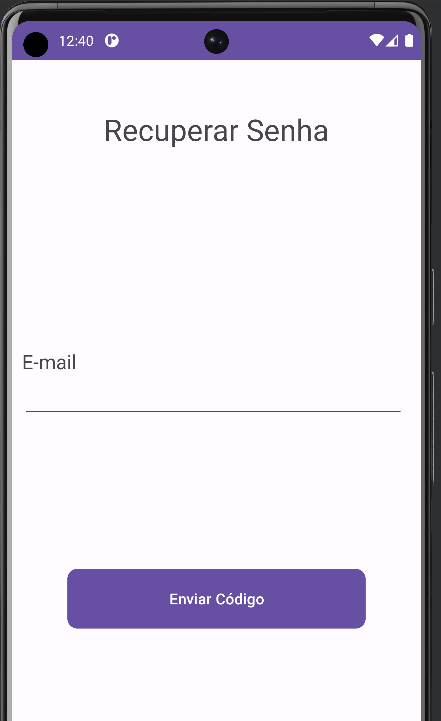


Código XML



3.1.5 Tela de recuperação de senha

* Essa tela foi projetada com o propósito de facilitar a recuperação de senha, permitindo que o usuário solicite uma nova senha com base apenas em seu endereço de e-mail.

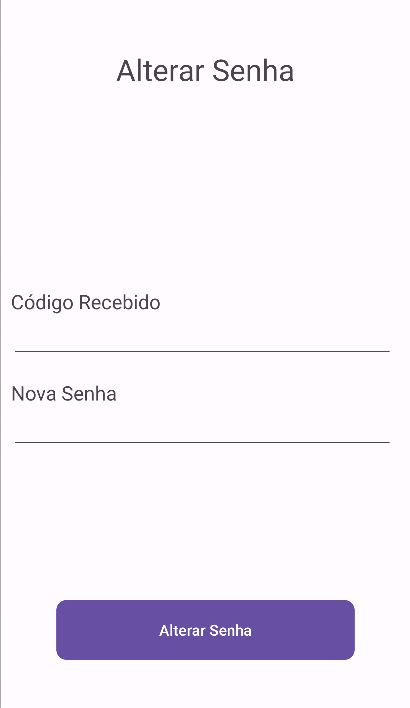


Código XML



3.1.6 Tela de alteração de senha

* Essa interface foi desenvolvida especificamente para a alteração de senha, permitindo que o usuário insira o código recebido para essa finalidade e defina uma nova senha para acessos futuros.



Código XML

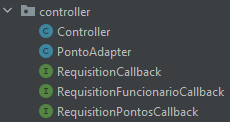


# ATIVIDADE 4 (30% da Nota Final)

## 4.1 Desenvolvimento

4.1.1 Desenvolvimento Front-end

Para o desenvolvimento do aplicativo Android, foi utilizada a linguagem JAVA, com o auxílio da IDE Android Studio. A arquitetura utilizada foi o padrão MVC (Model, View, Controller):



Package Controller

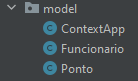
No package controller, foram colocadas as classes e interfaces responsáveis por fazer o meio de campo entre as telas do aplicativo e as regras de negócio.

**Classe Controller:** A classe mais importante do projeto, pois nela existem todas as funções responsáveis por se comunicar com a API .NET, fazendo as requisições assíncronas necessárias, retornando ou processando os resultados das mesmas.

**Classe PontoAdapter:** Essa classe é necessária para a listagem de pontos realizados que aparece na tela após o login do usuário, ela é responsável por receber os dados de um ponto, e transferimos para um card, que será detalhado posteriormente.

**Interfaces:** as 3 interfaces fazem a mesma coisa, porém com um retorno diferente cada uma. Elas têm o intuito de controlar a execução das requisições assíncronas que ocorrem, isso é necessário pois pelo fato de serem requisições assíncronas, elas ocorrem em paralelo ao resto do programa. Para que não ocorram falhas em casos que é necessário o retorno daquela requisição para continuar o programa, essa interface é responsável por aguardar esse retorno , evitando erros de threads no aplicativo.

A “RequisitionCallback’, apenas retorna um valor booleano, para o programa saber se a requisição funcionou ou não, a “RequisitionFuncionarioCallback” retorna um objeto do tipo funcionario, e o “RequisitionPontoCallback” retorna um array com todos os pontos registrados.

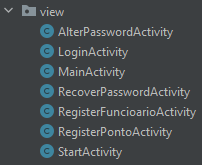


Package Model

No package model estão todas as classes referentes às regras de negócio do aplicativo.

ContextApp: nela estão alguns dados de contexto do aplicativo, como por exemplo: token JWT do usuário que está logado, para que não seja necessária a busca do token nas APIS a cada ação do usuário, mas sim a após a primeira busca desse token, já armazená lo localmente.

Funcionario e Ponto: essas são classes padrões para que se possa organizar os dados de cada um desses tipos em objetos para serem utilizados conforme o necessário.



Package View

Nesse package estão todas as partes lógicas das telas do aplicativo, nela se encontram todas as ações que devem ser realizadas após os cliques nos botões das activitys. Nelas estão todas as chamadas de requisições das classes do package controller, e também apresenta os objetos criados nas classes do package model.

Houveram algumas alterações em relação a primeira versão do projeto, como por exemplo, a alteração da cor padrão do aplicativo para verde, e também alguns ajustes de posicionamento dos elementos. Além disso, foi adicionada uma tela de início para melhor experiência do usuário, essa tela fica à mostra por 3 segundos e depois o usuário é direcionado a tela inicial:

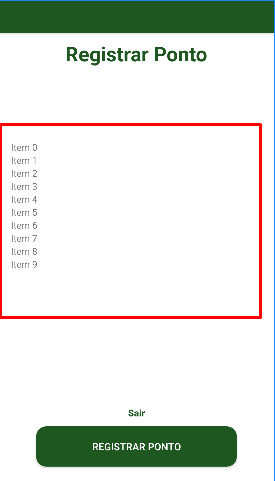


Tela Start

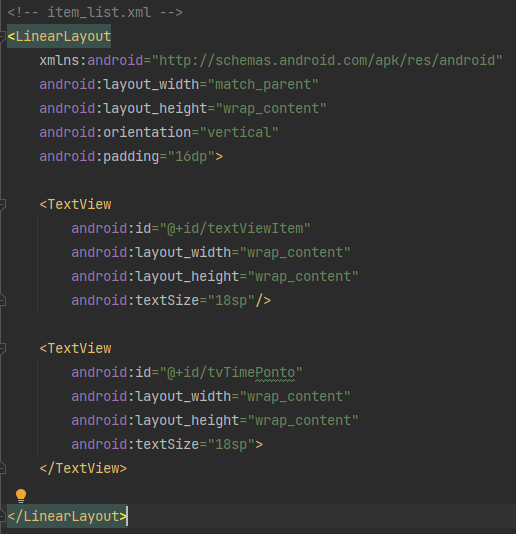


Código XML

Uma parte que não havia sido desenvolvida até o momento era a de listagem dos pontos na tela RegisterPontoAcitivity. Para isso, foi adicionado um Recyclerview a essa Activiy que é basicamente um espaço para listagem de itens.



Para representar cada item dessa listagem, foi criado um item\_list\_ponto, uma espécie de card, em que foram colocados cada elemento de um objeto ponto:



Código XML do item\_list\_ponto

Quando a tela RegisterPontoAcitivity é carregada, ela faz uma requisição pedindo à API os pontos registrados pelo usuário que está logado. A API retorna um array com esses pontos, esse array é enviado ao “PontoAdapter”, esse percorre o array, adicionado o os dados de cada objeto ponto em cada carview, assim criando uma listagem dinâmica e eficiente.

4.1.2 Desenvolvimento Back-end

Partindo da aplicação desenvolvida no último projeto integrador, que contemplava as funcionalidades de cadastro, login por JWT e recuperação de senha, desenvolvemos nesse projeto a funcionalidade de registro e busca de ponto eletrônico.

O repositório fica disponível em <https://github.com/lucassmann/PontoCalculatorAuth>.

Em questão de novas funcionalidades de ponto eletrônico, foram desenvolvidas novas rotas na API:

* Registro de ponto eletrônico:

[HttpPost("ponto")]

public IActionResult Register(string jwt, bool in\_out, string? details)

{

int userId;

try

{

var token = \_jwtService.Verify(jwt);

userId = int.Parse(token.Issuer);

}

catch (Exception ex){

return Unauthorized("Please login"); }

var ponto = new Ponto

{

User\_id = userId,

DateTime = DateTime.Now,

In\_out = in\_out,

Details = details

};

return Created(

"Success",

\_pontoRepository.Create(ponto)

);

}

* Busca de ponto eletrônico:

[HttpGet("ponto")]

public IActionResult Get(string jwt, int? pontoId = null, bool? today = null)

{

int userId;

try

{

var token = **\_jwtService**.Verify(jwt);

userId = int.Parse(token.Issuer);

}

catch (Exception ex) {

return Unauthorized("Please login");

}

List<Ponto> result =

**\_pontoRepository**.Get(

userId: userId,

pontoId: pontoId,

today: today);

return Ok(result);

}

Em ambos métodos, é utilizado o JWT de um usuário logado para delimitar o usuário e pontos. Funcionalidades para administradores não estão incluídas no escopo desse projeto.

Ainda, foi optado por utilizar parâmetros opcionais, de forma a possibilitar apenas 2 rotas para registro e busca do objeto ponto.

A classe do ponto é bem simples, utilizando do tipo DateTime do C#, assim como um booleano para representar entrada/saída do funcionário, de forma a padronizar e diminuir utilização de recursos.

public class Ponto

{

public int **Id** { get; set; }

public int **User\_id** { get; set; }

public DateTime **DateTime** { get; set; }

public required bool **In\_out** { get; set; }

public string? **Details** { get; set; }

}

No que toca à infraestrutura, foi optado por contanerizar a aplicação .Net e o banco de dados SQL Server, de forma a entender melhor a real utilização de recursos para ambos.

Para essa tarefa, foi primeiramente criado o arquivo docker-compose.yml para descrever os serviços tanto da aplicação quanto do banco de dados.

version: '3.4'

networks:

pontocalculatornetwork:

services:

pontocalculatorDB:

container\_name: pontocalculatorDB

image: mcr.microsoft.com/mssql/server:2022-latest

ports:

- 8002:1433

environment:

- ACCEPT\_EULA=Y

- MSSQL\_SA\_PASSWORD=S3nh4S3cr3t4!

networks:

- pontocalculatornetwork

deploy:

resources:

limits:

cpus: "1.0"

memory: 2000M

pontocalculator:

container\_name: pontocalculator

environment:

- ASPNETCORE\_ENVIRONMENT=Development

- ASPNETCORE\_URLS=http://+:80

ports:

- 8001:80

image: ${DOCKER\_REGISTRY-}pontocalculator

volumes:

- ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro

- ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro

build:

context: .

dockerfile: PontoCalculator/Dockerfile

depends\_on:

- pontocalculatorDB

networks:

- pontocalculatornetwork

deploy:

resources:

limits:

cpus: "1.0"

memory: 200M

Como se trata de uma aplicação teste e rodada localmente, as senhas foram deixadas explícitas no arquivo.

Para a versão final do desenvolvimento desse projeto, foram incluídas limitações de recursos de 1 núcleo lógico de processamento para cada contêiner, assim como 200 megabytes de memória para a aplicação .Net e 2000 megabytes de memória para o banco de dados, segundo os requisitos mínimos apontados pela Microsoft.

Para a criação do Dockerfile, a própria IDE Visual Studio já faz a geração automática através das opções de configuração na interface.

FROM mcr.microsoft.com/dotnet/aspnet:**7.0** AS base

WORKDIR /app

EXPOSE 8001:80

FROM mcr.microsoft.com/dotnet/sdk:**7.0** AS build

WORKDIR /src

COPY ["PontoCalculator/PontoCalculator.csproj", "PontoCalculator/"]

RUN dotnet restore "PontoCalculator/PontoCalculator.csproj"

COPY . .

WORKDIR "/src/PontoCalculator"

RUN dotnet build "PontoCalculator.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "PontoCalculator.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final

WORKDIR /app

COPY --from=**publish** /app/publish .

ENTRYPOINT ["dotnet", "PontoCalculator.dll"]

Para assegurar que a aplicação sempre tenha uma inicialização de sucesso durante os testes, foi necessário explicitar as portas a serem expostas, de forma que a aplicação e banco de dados utilize portas fixas. Isso porque a aplicação .Net precisa ser configurada para acessar o banco de dados SQL server, assim como a aplicação frontend Android precisa ser configurada para acessar os endpoints da aplicação .Net rodando localmente.

# ATIVIDADE 5 (30% da Nota Final)

## 5.1 Apresentação de resultados

5.1.1 Apresentação da funcionalidade da aplicação

Para apresentar os resultados, ou seja, o aplicativo em funcionamento, achamos interessante gravar a tela de um celular, com o aplicativo rodando, simulando um cenário real. No vídeo disponibilizado pelo link abaixo, foi realizado um cadastro de usuário, e logo após, o login. Na tela de registro de ponto podemos ver que não existe nenhum ponto registrado, mas após os cliques no botão de registro, os mesmo já são listados automaticamente.

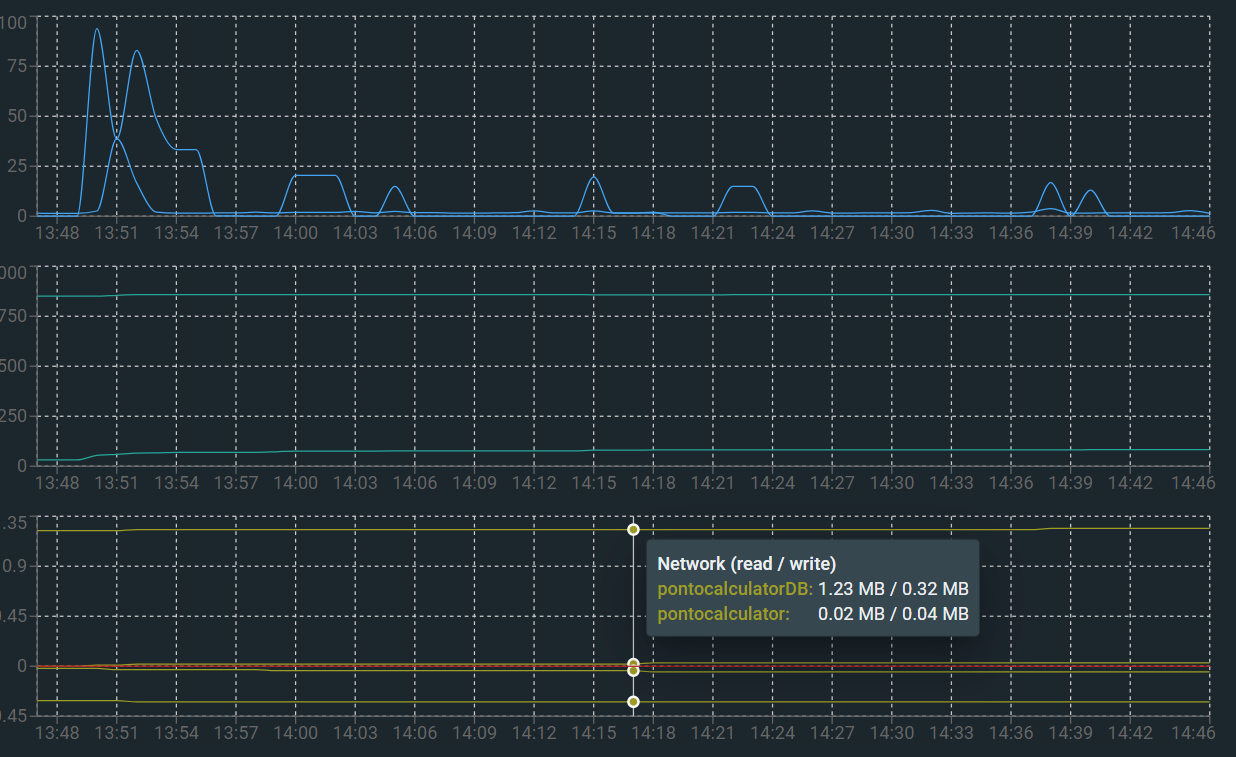
Link: <https://youtu.be/PWB6To46EWg>

5.1.2 Apresentação da análise de infraestrutura back-end

Para medir a infraestrutura necessária e utilização de recursos, foi realizada uma nova medição utilizando a extensão Live Charts, disponibilizada no Docker Desktop.

Rota de autenticação:

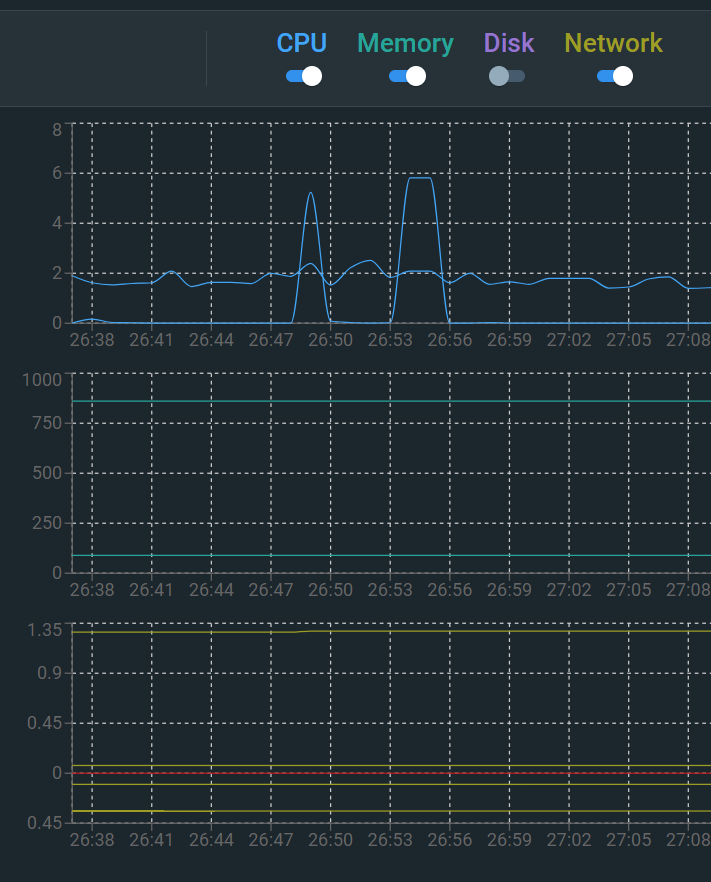
1. POST register: registra o usuário no banco de dados, com os parâmetros email, senha e nome.
2. POST login: faz o login do usuário, com parâmetros de email e senha, retornando um JWT.
3. GET user: retorna os dados de usuário com omissão do dash da senha, utilizando o JWT como parâmetro.
4. POST forgot-password: registra um token de recuperação de senha na tabela do usuário no banco de dados e envia um email de recuperação de senha.
5. POST reset-password: usa o token de recuperação de senha para redefinir uma nova senha.



Aqui vemos que o registro de usuário é a chamada mais intensiva em consumo de recursos de processamento, assim como notamos um aumento da memória usada nesse momento.

Rota de ponto eletrônico:

1. POST ponto
2. GET ponto



Na rota de ponto eletrônico vemos que o consumo de processamento da aplicação ultrapassa o consumo de processamento do banco de dados quando uma requisição é realizada. Já o consumo de memória e rede se manteve sem variação considerável.

Ainda, uma hipótese formulada foi que a chamada GET ponto quando configurado para todos os pontos pode consumir mais processamento que POST ponto, devido atuar em um número múltiplo de registros.

Ao comparar os testes no começo e final do desenvolvimento desse projeto, notamos que a configuração de um uso limitado de recursos de núcleos de processamento e memória tem impacto considerável. Enquanto na primeira medição o banco de dados SQL Server sempre teve medições bastante acima da aplicação .Net, na segunda medição vemos que o consumo de processamento do banco de dados teve um aumento menos pronunciado no uso ativo de operações, enquanto a aplicação .Net teve consumo de processamento bastante variável na medida em que as chamadas eram feitas.

O teste foi realizado em um processador AMD Ryzen 5 3550H, com limitação de 1 núcleo lógico para cada container. O uso máximo beirou um consumo de 6% de 1 núcleo para a aplicação e 2% de um núcleo para o banco de dados. Assim, concluímos que existe a possibilidade de reduzir consideravelmente o limite de uso de processamento, de forma que cada container utilize menos de 0.1% de núcleo do mesmo modelo de processador.

No que diz respeito ao consumo de memória, a limitação de recursos da aplicação e do banco de dados, respectivamente em 200 megabytes e 2000 megabytes, não teve impacto significativo no uso de recursos de memória. Isso já era esperado, visto que na primeira medição já não foram atingidos picos de uso maiores que as limitações impostas.

Visto os resultados dos dois testes, os requisitos de infraestrutura mínimos para o sistema se mantêm inalterados para a memória, mas reduzem consideravelmente para o processamento. Ainda, essa redução de recursos de processamento resultou em um impacto aceitável na velocidade de resposta das requisições, na faixa 3 a 8 segundos para o primeiro teste e 3 a 10 segundos para o segundo teste, tendo o registro de usuário como a chamada de maior delay de resposta no segundo teste e a recuperação de senha no primeiro teste, ambos casos de uso não frequentes quando comparados à rota de ponto eletrônico e ao login de usuário.