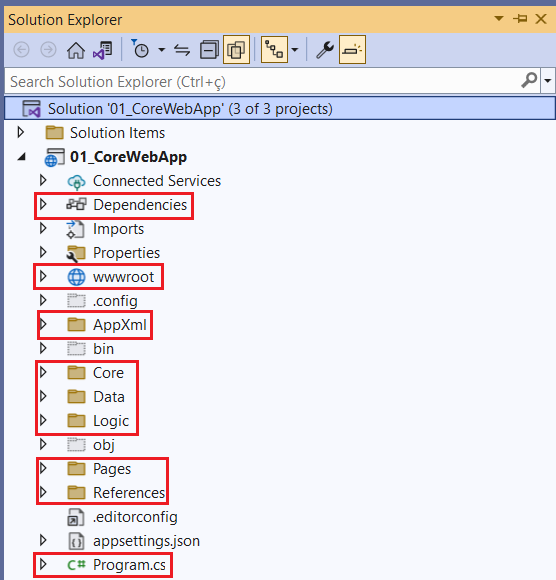
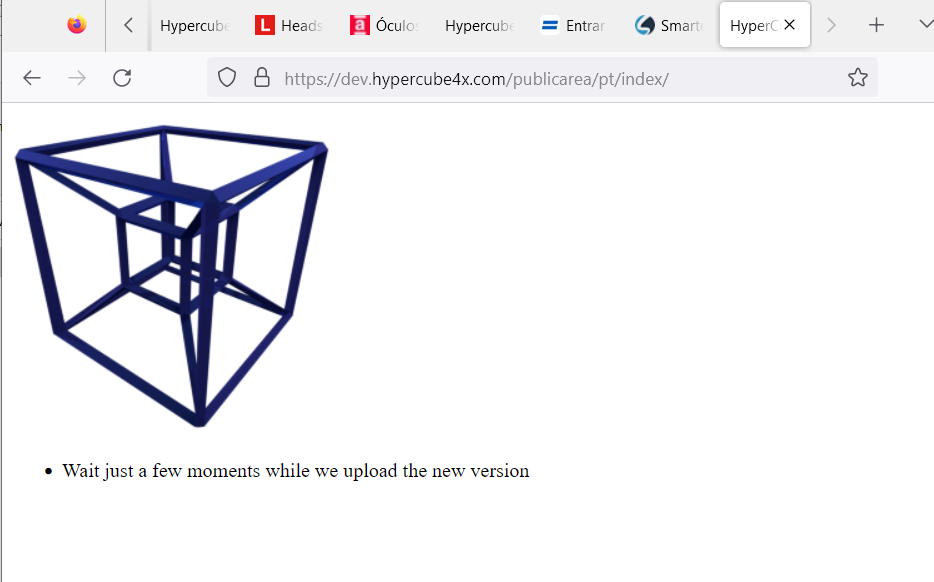
# Estrutura da Aplicação



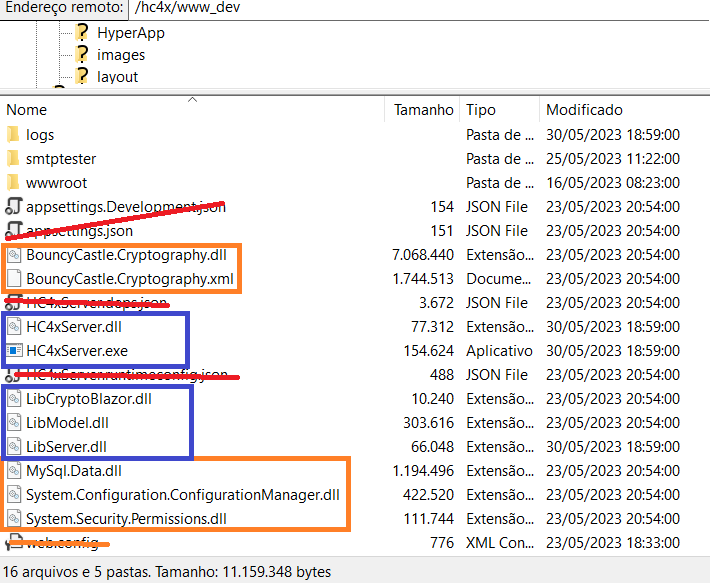
* Program.cs – programa inicial do projeto (void main)
* Dependencies / References – Na pasta references temos os arquivos: BouncyCastle.Crypto.dll, LibCryptoBlazor.dll, LibModelBlazor.dll e HyperServerBlazor.dll. Os arquivos LibModelBlazor.dll e HyperServerBlazor.dll devem ser referenciados usando Dependencies, Add COM reference...
* Wwwroot – São os arquivos estáticos, como .css, .js e imagens
* AppXml – O arquivo HyperCube.xml contém as interfaces html que serão usadas pelo site. Ver o nó SectorInterface.
* Core – Classes do namespace hc4x\_WebApp.Core, responsáveis pela abstração de processos do asp.net / razor.
* Data – Arquivos .xml para teste de manipulação de dados.
* Logic – Código fonte principal. Nele serão implementadas as rotinas do website.

# Atualização de Versão

Colocar o arquivo app\_offline.htm na raiz do site e aguardar até que o site exiba seu conteúdo.



Subir apenas os arquivos executáveis da pasta bin\release\net7.0 da versão nova (.dll e .exe) na raiz. Subir os arquivos em AppXml e em wwwroot.



Remover o arquivo app\_offline.htm e testar o site.

# Routes e áreas do site

O site é composto por três áreas:

publicarea – conjunto de páginas acessível por qualquer internauta (anônimo)

privatearea – conjunto de páginas acessível por usuários cadastrados

rest – comunicação com o hypercube client. Retorna arquivos .xml, .json, imagens, entre outros

As rotas são formadas com a seguinte sintaxe:

{area}/{lang}/{pageid}

* area – uma das três opções: publicarea, privatearea, rest
* lang – pt (português), em (inglês)
* pageid – uma das interfaces cadastradas em hypercube.xml. Se a interface não estiver cadastrada, será apresentada mensagem de erro.

# Arquitetura do sistema (Classes)

[Fazendo ainda]

# How To

ndRoute.FullPublicRoute("confirm-email", "18;124a4078", true)

18 => Chave primária do usuário

124a4078 => atName.substring(0, 8)

Saída:

http://localhost:5185/publicarea/pt/confirm-email/secureqs=44664762574b657572643636544874396147304b57773d3d

private string QueryDecrypt(string parQS) {

int iSecureqs;

string retValue;

try {

if ((retValue = parQS).Length == 0) return (retValue);

iSecureqs = retValue.IndexOf(c\_secureqs);

if (iSecureqs > -1) {

retValue = retValue.Substring(iSecureqs + c\_secureqs.Length + 1);

retValue = GearAux.UTF8String(GearAux.ParseByte(retValue));

retValue = axSession.DecryptUTF8(retValue);

}

}

catch (Exception Err) { retValue = string.Empty; axMundi.ShowException(Err, Name, nameof(QueryDecrypt)); }

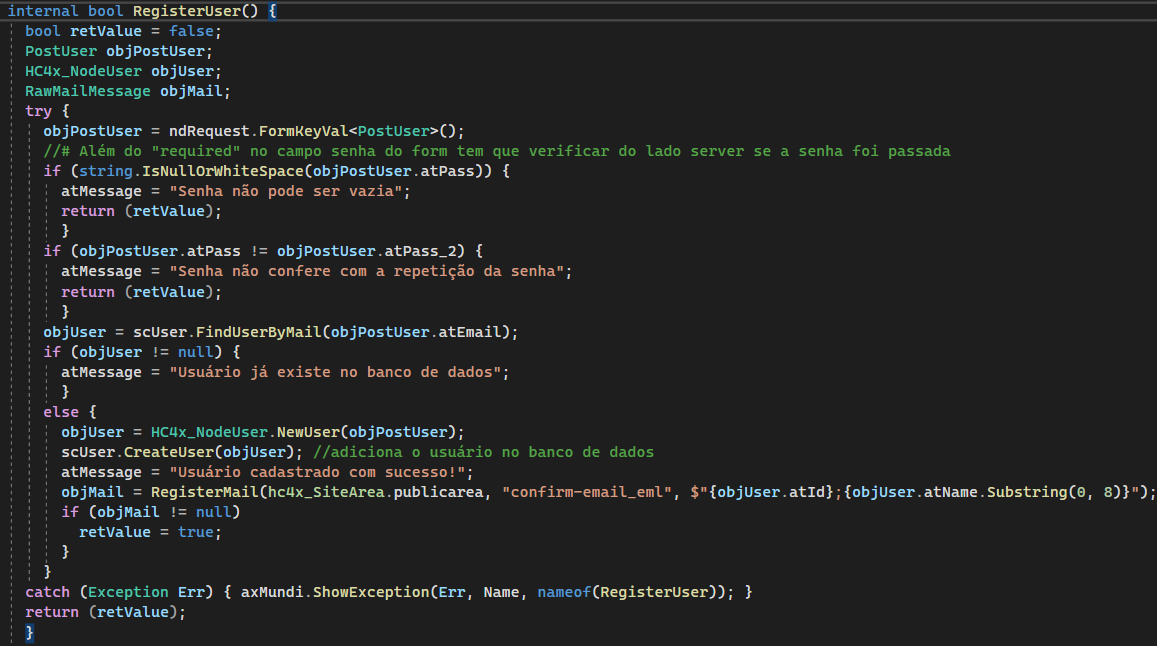
return (retValue);

}

# Confirmar email

Quando o usuário se registra no site, é chamado o método RegisterUser().

Método RegisterUser abaixo:



O método verifica se as informações passadas nos inputs estão corretas de acordo com as validações referentes respectivamente: senha do usuário em nula ou em branco, senha do usuário e confirmação de senha são iguais, verificação se o usuário (email) existe no banco.

Caso o usuário não exista, ele é criado e adicionado ao banco de dados com as seguintes linhas do código:

objUser = HC4x\_NodeUser.NewUser(objPostUser); // criando o usuário

scUser.CreateUser(objUser); //adicionando o usuário criado ao banco de dados.

atMessage = "Usuário cadastrado com sucesso!"; // mensagem de sucesso para criação e inclusão do usuário.

Após essas operações, é chamado o método RegisterMail() para criar o email de envio para o usuário. O retorno desse método é atribuído a variável objMail que é do tipo RawMailMessage.

objMail = RegisterMail(hc4x\_SiteArea.publicarea, "confirm-email\_eml", $"{objUser.atId};{objUser.atName.Substring(0, 8)}");//criar o email para mandar para o usuário.

OBS: os parâmetros do método RegisterMail são:

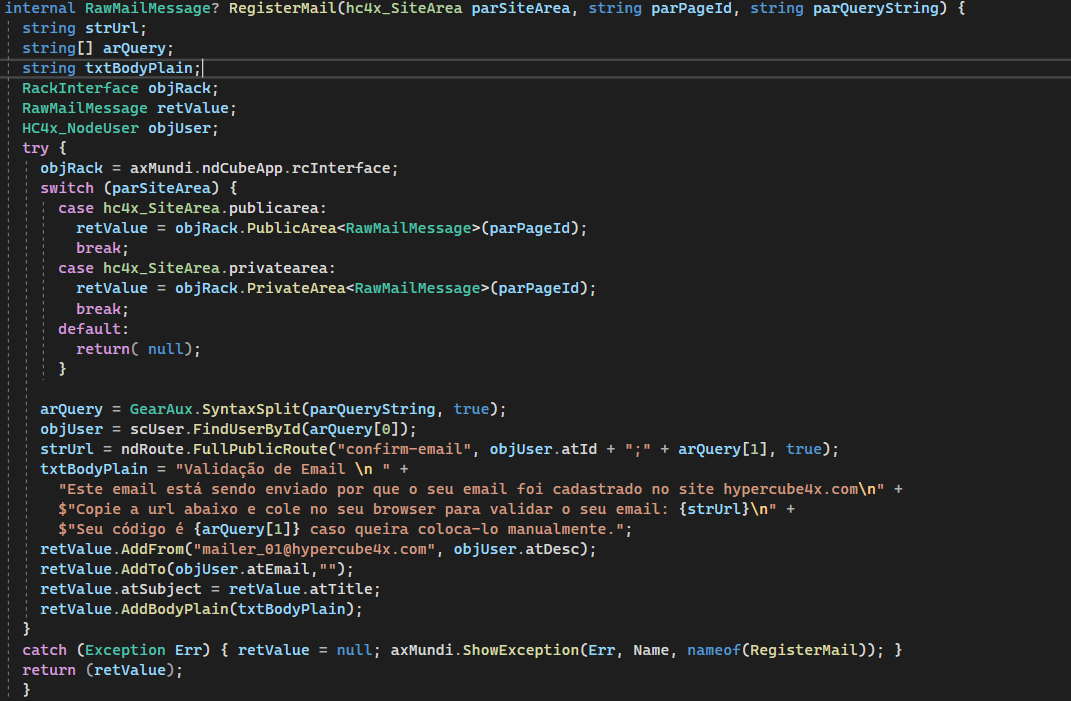
RegisterMail(hc4x\_SiteArea parSiteArea, string parPageId, string parQueryString)

1° Área do site (pública ou privada)

2° Página que será redirecionado o usuário

3° Id do usuário;Código do usuário **É DE EXTREMA IMPORTÂNCIA O “;” PARA DELIMITADOR ENTRE O ID E O CÓDIGO PARA FAZER A SEPARAÇÃO POSTERIORMENTE NA STRING COM O MÉTODO GearAux.SyntaxSplit().**

Método RegisterMail abaixo:



A finalidade desse método é gerar a url com o código para validar o email do usuário e montar o email que será enviado para o usuário.

arQuery = GearAux.SyntaxSplit(parQueryString, true); //separação do Id do usuário e do código de confirmação de email

objUser = scUser.FindUserById(arQuery[0]); //verifica se o usuário existe no banco de dados

strUrl = ndRoute.FullPublicRoute("confirm-email", objUser.atId + ";" + arQuery[1], true); // cria a Url que irá para o email do usuário posteriormente

OBS:

O método ndRoute.FullPublicRoute() requer três parâmetros sendo eles respectivamente:

1° id da página confirmar email

2° Id do usuário e código de confirmação separados por ponto e vírgula

3° Flag para requisição de criptografia ou não (true se sim, false se não)

modelo de url criado = <http://localhost:5185/publicarea/pt/confirm-email/secureqs=566979696c36614f4d75415769706934486e4c4d77773d3d>

Criando o texto plano para mandar para o email do usuário:

txtBodyPlain = "Validação de Email \n " +

"Este email está sendo enviado por que o seu email foi cadastrado no site hypercube4x.com\n" +

$"Copie a url abaixo e cole no seu browser para validar o seu email: {strUrl}\n" +

$"Seu código é {arQuery[1]} caso queira coloca-lo manualmente.";

Adicionando respectivamente:

1° AddFrom: retValue.AddFrom("mailer\_01@hypercube4x.com", objUser.atDesc);

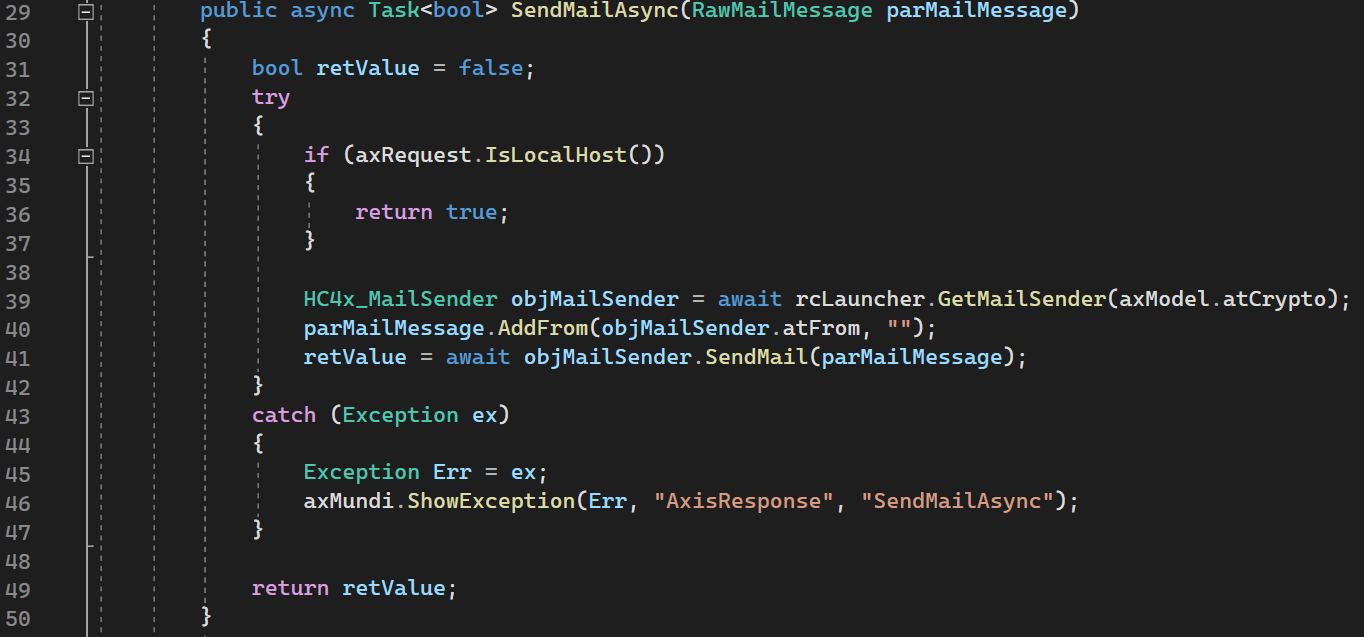
2° AddTo: retValue.AddTo(objUser.atEmail,"");

3° atSubject: retValue.atSubject = retValue.atTitle;

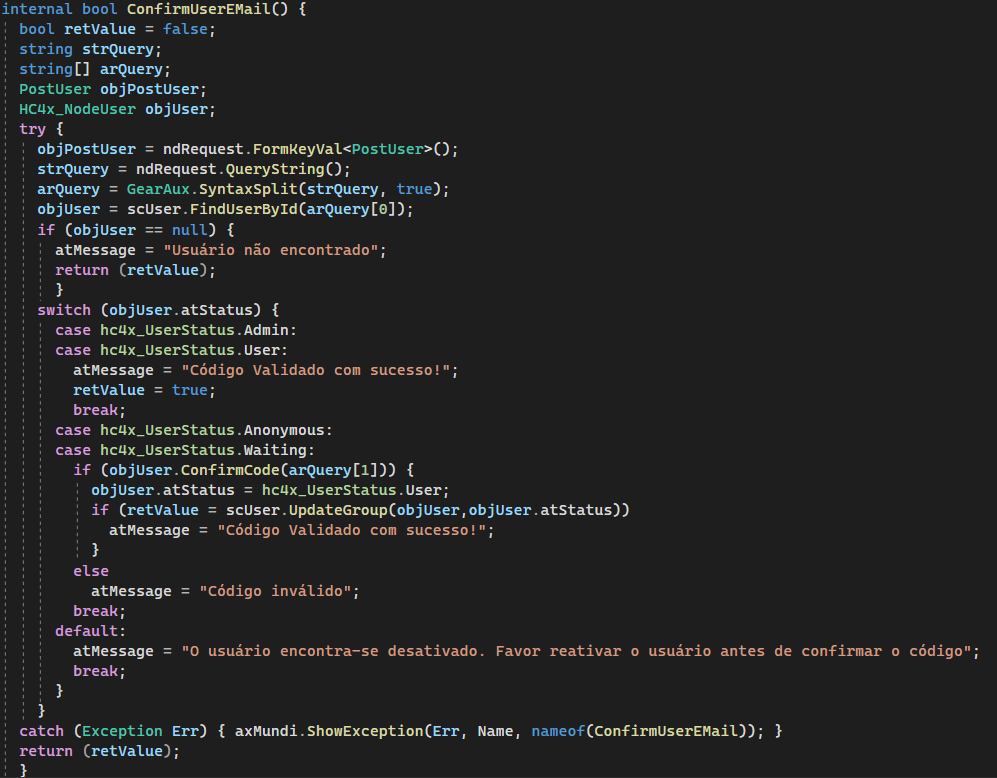
4° AddBodyPlain: retValue.AddBodyPlain(txtBodyPlain);

Após o usuário clicar na url que estará no email, será executado o método ConfirmUserEMail()

**OBS: O código só é enviado ao email do usuário se estiver fora do localhost, ou seja, se estiver testando e implementando uma nova funcionalidade que requeira o uso do envio de email, só irá funcionar se o código estiver ativo no ambiente de dev/test/produção.**

****

Método ConfirmUserEMail abaixo:



A finalidade do método é trocar o status do usuário de Waiting para User.

Primeiramente, o usuário é procurado no banco de dados. Caso ele não exista, é retornada a mensagem “Usuário não encontrado” e o retorno do método é falso;

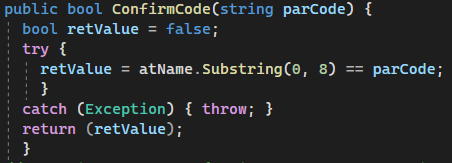
if (objUser == null) {

atMessage = "Usuário não encontrado";

return (retValue);

}

Caso o usuário exista, é verificado o status do usuário (Status do usuário dita se ele poderá ou não acessar a parte interna do site, podendo acessar somente no caso User ou Admin) e caso esteja como Waiting, é verificado se o código passado é o mesmo código mandado para o email com o método ConfirmCode.



public bool ConfirmCode(string parCode) {

bool retValue = false;

try {

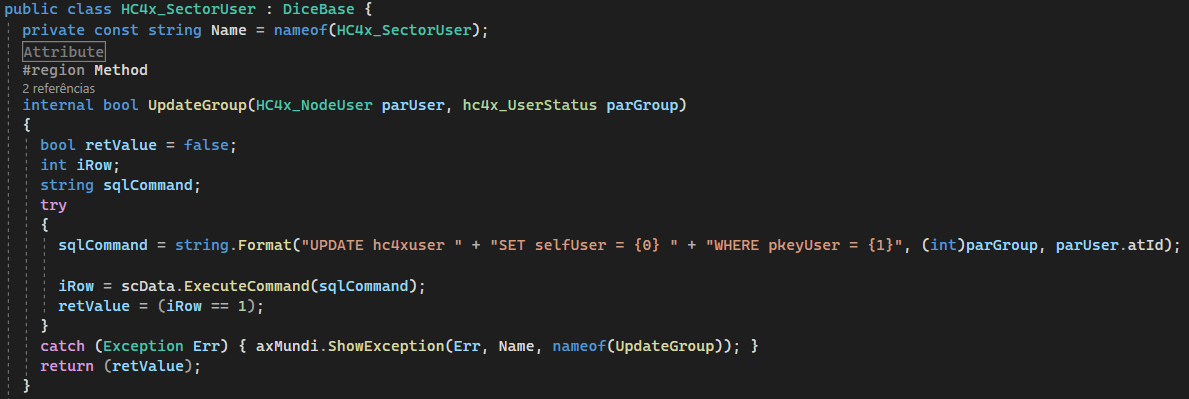
retValue = atName.Substring(0, 8) == parCode;

}

catch (Exception) { throw; }

return (retValue);

}



Caso o código esteja correto, é invocado o método UpdateGroup cuja finalidade é atualizar o status do usuário no banco para que ele possa acessar o site. Caso dê certo, o método retornará true.

internal bool UpdateGroup(HC4x\_NodeUser parUser, hc4x\_UserStatus parGroup)

{

bool retValue = false;

int iRow;

string sqlCommand;

try

{

sqlCommand = string.Format("UPDATE hc4xuser " + "SET selfUser = {0} " + "WHERE pkeyUser = {1}", (int)parGroup, parUser.atId);

iRow = scData.ExecuteCommand(sqlCommand);

retValue = (iRow == 1);

}

catch (Exception Err) { axMundi.ShowException(Err, Name, nameof(UpdateGroup)); }

return (retValue);

}