**GIT e GITHUB**

***Prompt de Comando (CMD)***

*COMANDOS*

***dir*** - Lista todas as pastas e arquivos do diretório atual. Por exemplo, ao entrarmos no CMD ele entra no diretório *users/elias*, então ao digitarmos *dir* e dar enter, ele vai mostrar todos os arquivos da pasta elias.

PS.: no Linux o ***ls***  equivale ao ***dir***.

***cd*** - Usado para navegar entre pastas. ***cd /*** por exemplo, nos faz entrar no diretório *C:* Estando no diretório C, se quisermos entrar na pasta Windows, é só digitar ***cd Windows***. E se quisermos voltar para o diretório anterior, no caso C, usamos o comando ***cd ..*** (cd dois pontos).

PS.: no Linux também usamos ***cd***.

***cls*** - Limpa a tela. No Linux é o ***clear*** ou ***crtl + L***.

***Tecla Tab*** - Serve para completar nomes de arquivos, pastas etc. Por exemplo, se estivermos no diretório ***C***, dentro tem a pasta Windows, então se quisermos acessar essa pasta, basta digitarmos ***cd*** ***W*** e teclarmos ***Tab***.

***mkdir (make directory)*** - Usado para criar diretórios. Para criarmos uma pasta, basta digitarmos ***mkdir nomeDaPasta***.

***del***  - Deleta todos os arquivos de um repositório (pasta). Por exemplo, usamos ***mkdir workspace*** e criamos essa pasta; entramos na nova pasta usando ***cd workspace***; e já dentro da pasta digitamos ***echo hello > hello.txt*** (*echo* imprime na tela o que digitarmos depois dele, e o *> hello.txt*  cria um arquivo txt com o que foi impresso por *echo*). Então para deletar o conteúdo da pasta *workspace* nós voltamos para o diretório C usando ***cd ..*** e digitamos ***del workspace***, assim o arquivo txt desta pasta será deletado.

***rmdir*** - Deleta a pasta e todo o seu conteúdo. Então, estando em *C*, digitamos ***rmdir workspace /S /Q*** e a pasta será deletada. As flags /S /Q servem para não pedir confirmação.

PS.: no Linux o comando é ***rm*** e as flags para apagar todas as pasta dentro de uma pasta é ***-rf*** o *r*  é de recursivo, ou seja, para todas as pastas, e o *f* é para que não peça confirmação. Então o comando ficaria assim ***rm -rf workspace***. Com isso a pasta workspace e todas as suas subpastas seriam apagadas.

***Git***

O Git é um sistema distribuído e seguro. Distribuído porque a versão final de um código que fica armazenado na nuvem também está armazenada no computador do seu criado e ainda pode estar nos computadores de outros possíveis colaboradores. E é segura porque se houver algum problema em um ou vários computadores dos colaboradores, ele ainda estará na nuvem. E ainda que dê um problema na nuvem, haverá ainda o código no computador de algum colaborador.

***Funcionamento do Git por baixo dos panos***

***SHA1*** - SHA significa Secure Hash Algorithm (Algoritmo de Hash Seguro), ele nada mais é que um algoritmo que gera uma chave hash a partir do seu arquivo original. Esta chave hash é forma por 40 caracteres. Esse Algoritmo foi desenvolvido pela Agência de Segurança Nacional do EUA (NSA).

Para criar uma chave hash de um arquivo específico podemos entrar na pasta onde se encontra o arquivo, clicar com o botão direito e clicar em Git Bash. Vai ser aberto uma janela como de um terminal nele é só digitar o comando ***openssl sha1 nomeDoArquivo.extensão*** e assim será gerada uma chave hash que aparecerá no teminal.

***Tipo de Objetos do Git***

***Blob*** - As blobs são os campos básicos de composição. Os objetos deste tipo tem um metadado (cabeçalho) que segue esse padrão:

*Blob*

*9 //Tamanho do arquivo*

\0

conteúdo

***Tree*** - Já as *trees* armazenam vários *blobs* e *comits* e apontam para eles ou até mesmo para outras árvores (*trees*), afinal um diretório pode conter outros diretórios, e como as *trees* indicam também a localização dos arquivos, faz sentido que elas apontem para outras árvores. As árvores, assim como as *blobs*, também contém sha1, que são alterados quando qualquer arquivo é alterado. As *trees* também tem metadados e eles seguem esse padrão:

*Tree*

*\0*

*Blob 4sc543c (sha1) texto.txt*

***Commit*** - O principal objeto do *Git*, pois é ele que contém o autor e a mensagem referente ao(s) arquivo(s) que estão sendo (ou foram) produzidos. Além disso ele tem um sha1 que garante a confiabilidade, pois qualquer alteração em algum dos arquivos alterará o sha1 completamente. Isso se dá porque o *commit* aponta para todo o conjunto de árvores, ao apontar para a que está no topo da estrutura de dados, sendo assim, a mínima alteração em um arquivo será passada pelas árvores até chegar no *commit*. O metadado do *commit* segue esse padrão:

*Commit*

*Tree 54dsd45e*

*Parente b89c8d8d*

*Autor Parkles*

*Mensagem “Inicia...”*

*Timestap //Marca em milissegundos, que partem de 1970, o tempo em que foi criado.*

***Criando chaves SSH no GitHub***

**1º** **Na sua conta do GitHub, vá em** Settings >>> SSH and GPG keys >>> New SSH.

**2º No Git Bash digite:** ssh-keygen -t ed25519 -C [seu@email.com](mailto:seu@email.com) >>> Enter >>> Enter >>> Crie a senha da sua chave privada.

**3º** **Ainda no Git Bash, entre na pasta em que foi criadas as chaves SSH:**

Cd /c/Users/nome/.ssh/ >>> ls *(para listar o conteúdo do diretório)* >>>

Cat id\_ed25519.pub (*esse comando vai nos dá a chave pública criada)* >>> Copiamos a chave criada junto com email e então colamos no campo indicado no site do GitHub.

**4º Voltamos ao GitBash ainda na pasta .ssh e digitamos:** eval $(ssh-agent -s) (*isso irá criar um agente que irá passar nossa chave privada para o site do GitHub sempre que nos conectarmos) >>>* ls >>> ssh-add id\_25519 *(aqui irá pedir nossa senha e assim nossa chave privada será passada para o GitHub)*

Depois de configurada a chave SSH, para ***clonar um repositório privado*** presente no GitHub devemos ir no botão verde CODE >>> Clone SSH >>> E copiamos esse link >>> Vamos agora no GitBash instalado no PC, entramos na pasta em que pretendemos baixar os arquivos clonados, e digitamos:  ***git clone*** *colamos o link SSH >>>* Ele vai perguntar se estamos certo disso e então digitamos *yes.*

Podemos também usar ***tokens*** para ter acesso a nossos repositórios. Para criar um *token* devemos ir em Settings >>> Developer Settings >>> Personal access tokens >>> Generate new token >>> Damos um nome, um tempo para expiração, selecionamos *repo* e, enfim, clicamos em *Generate Token.*

Para clonar um repositório usando o token nós vamos em Code >>> Copiamos o endereço HTTPS >>> Entramos no GitBash, e por ele acessamos a pasta onde será baixado o repositório >>> Digitamos

***git clone*** *Colamos o endereço https* >>> Vai ser pedido o token e aí é só colar o seu token e o repositório será clonado.

**Ini*ciando o Git e criando um commit***

***git init*** - Para iniciar o *Git* nós vamos até a pasta onde queremos inicializa-lo, abrimos o *GitBash* com o botão direito, e então digitamos o comando ***git init***. O G*it* irá criar uma pasta oculta chamada *.git*. Ela é oculta porque é uma pasta gerencial, onde fica todo o conteúdo do *Git*, com os objetos que estaremos trabalhando. Para visualiza-la teremos de usar a flag *-a* ao lado do comando listar, assim: ***ls -a***.

***git config*** - Na primeira vez que usamos o *Git,* ele exige algumas configurações para que ao criarmos um commit ele tenha um autor. E para isso devemos cadastrar nele um email. Assim devemos usar o comando no GitBash: ***git config --global user.email [“seu@email.com”](mailto:\“seu@email.com\”)***

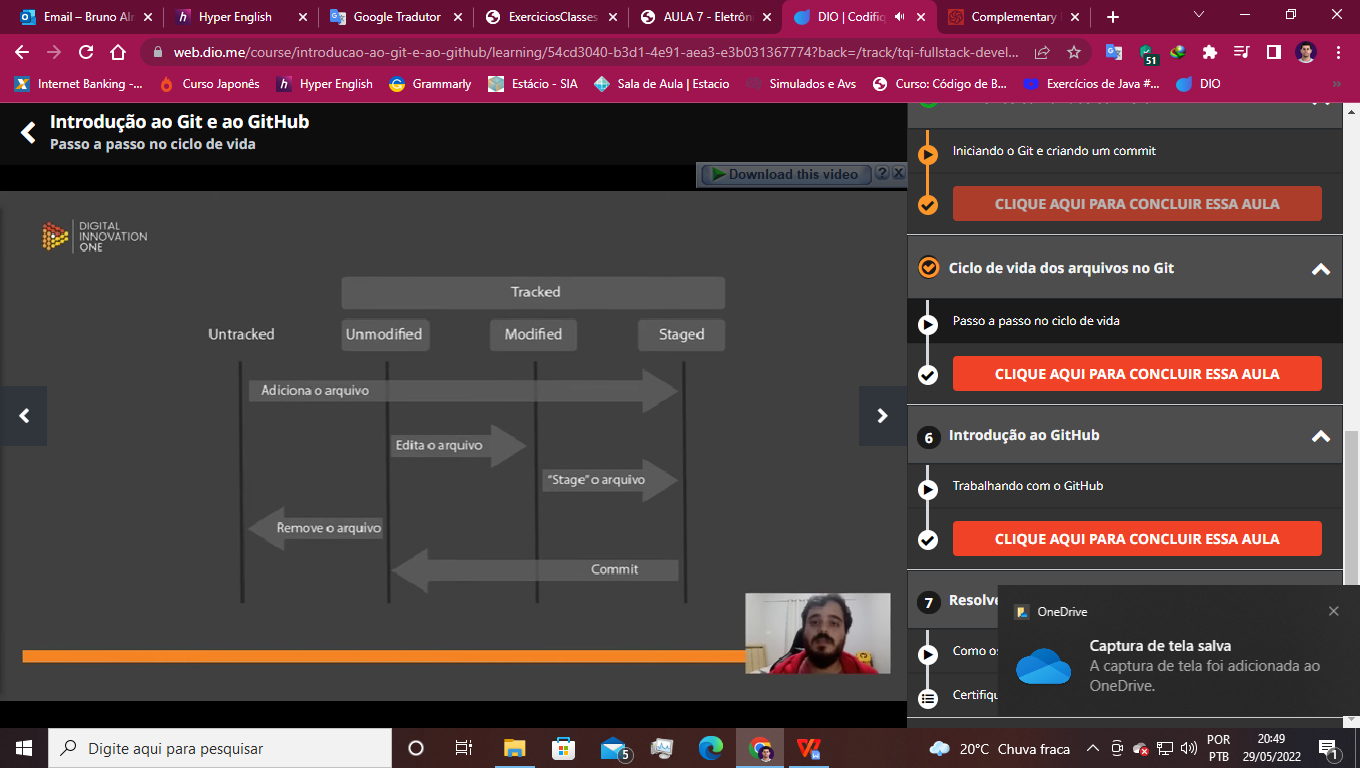
E depois de cadastrado o email, devemos cadastrar o nome de usuário, assim: ***git config --global user.name SeuNome***

***Criando o commit*** - Quando criamos um arquivo e salvamos dentro da pasta que contem o repositório *.git*, então devemos criar o commit que irá conter as informações sobre o nosso arquivo (devemos criar um commit também quando fizermos alterações significativas em nosso arquivo). Para isso, com GitBash entramos na pasta que contem o arquivo e adicionamos um commit com o comando ***git add \**** (o \* leva todas as alterações para *staging area*), e depois com ***git commit -m “mensagem inicial”*** criamos de fato o commit, sendo a flag *-m* usada para indicar um commit e a mensagem entra aspas vai trazer informações sobre o nosso arquivo, como alterações ou do que ele se trata.

***Ciclo de vida dos arquivos***

Em relação ao git os arquivos podem ser divididos em dois:

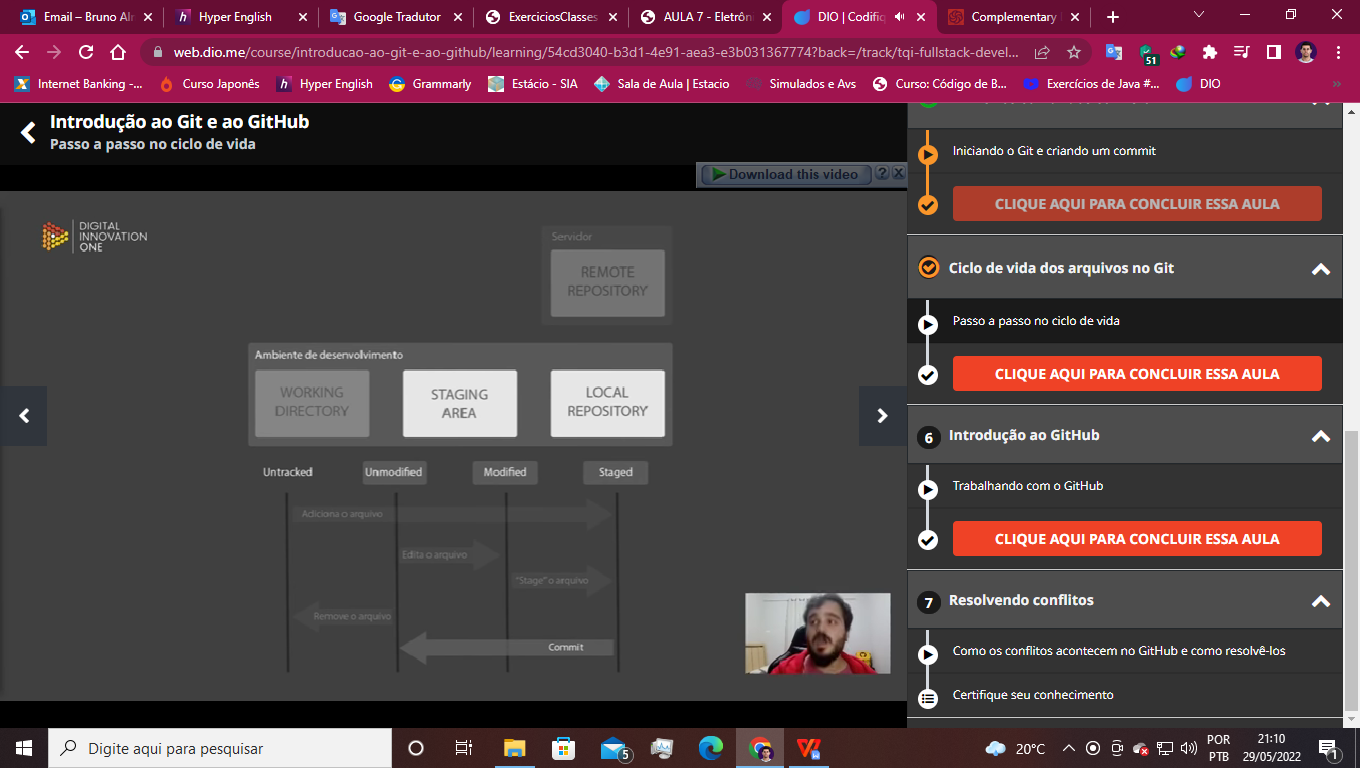
* ***Tracked*** - arquivos que o git **tem** conhecimento;
* ***Untracked*** - arquivos que o git **não tem** conhecimento.



Os arquivos listados como *tracked* podem ser dos tipos:

* ***Unmodified*** - Que não sofreram modificações;
* ***Modified*** - Que sofreram alguma modificação;
* ***Staged*** - Stage significa palco. Quando falamos de um arquivo staged, estamos falando de arquivo que está prestes a encenar, e essa encenação consiste em fazer parte de outro grupo. Por exemplo, quando damos o comando ***git add*** nós transferimos o arquivo de *modified* para *staged*, pois agora foi adicionado um commit e o arquivo passa a aguardar o comando ***git commit*** para se tornar de fato um commit e voltar a fazer parte do grupo *unmodified*.

Quando estamos trabalhando, incluindo e modificando arquivos no nosso diretório, nós estamos trabalhando no nosso *Working Directory*. Quando adicionamos um commit com o *git add,* nós estamos transferidos esses arquivos para *Staging Area* e assim o colocamos atrás do palco, preparando-os para entrar em cena. E quando de fato criamos o commit com o comando *git commit*, nós pomos os arquivos no palco, e esse palco se chama *Local Repository*, que é onde os arquivos são salvos e preparados para serem enviados para o *Remote Repository*, criando, dessa forma, uma imagem dos nossos arquivos com suas últimas alterações, e essa “fotografia” é que vai nos garantir segurança quanto a integridade desses dados.



Para verificarmos em qual desses status os nossos arquivos estão usamos, no GitBrash, o comando ***git status***.

Para mover um arquivo no GitBash usamos o comando:

***mv arquivo.ar ./outraPasta/***

***GITHUB***

Para que a configuração da sua conta no GitHub esteja de acordo com a configuração do Git do seu repositório é necessário entrarmos no GitBash e digitarmos ***git config --list***, isso irá listar as configurações do seu Git, e o principal é ver se o username e o email são os mesmos cadastrados no GitHub, pois, caso contrário, o Git na sua máquina irá criar os commits com uma assinatura diferente dos criados no GitHub, já que em um terá um usuário e email diferente do outro.

Se o email estiver diferente, devemos usar o comando:

***git config --global --unset user.email***

Isso irá apagar o email cadastrado. E se o username for diferente usamos:

***git config --global --unset user.name***

Para apagar o username do Git.

Para cadastrar um novo email e um novo username usamos os comandos:

***git config --global user.email [“seu@email.com”](mailto:\“seu@email.com\”)***

***git config --global user.name SeuNickName***

***Conectando um repositório local a um remoto***

Para conectar um repositório local a um repositório do GitHub devemos criar um repositório no GitHub, copiar o link que vai ser fornecido para este repositório e então abrir o GitBash no endereço do seu repositório local (onde estão seus arquivos) e digitar:

***git remote add origin https:linkCriadoNoGitHub.com***

Para listar os repositórios remotos conectados ao nosso repositório local usamos o comando:

***git remote -v***

Ele vai mostrar o endereço do repositório remoto, mas antes desse link vamos ver a palavra ***origin*** que é o nome padrão dado aos repositórios.

Depois disso, é só “empurrar” o repositório local para o remoto usando o comando:

***git push origin master***

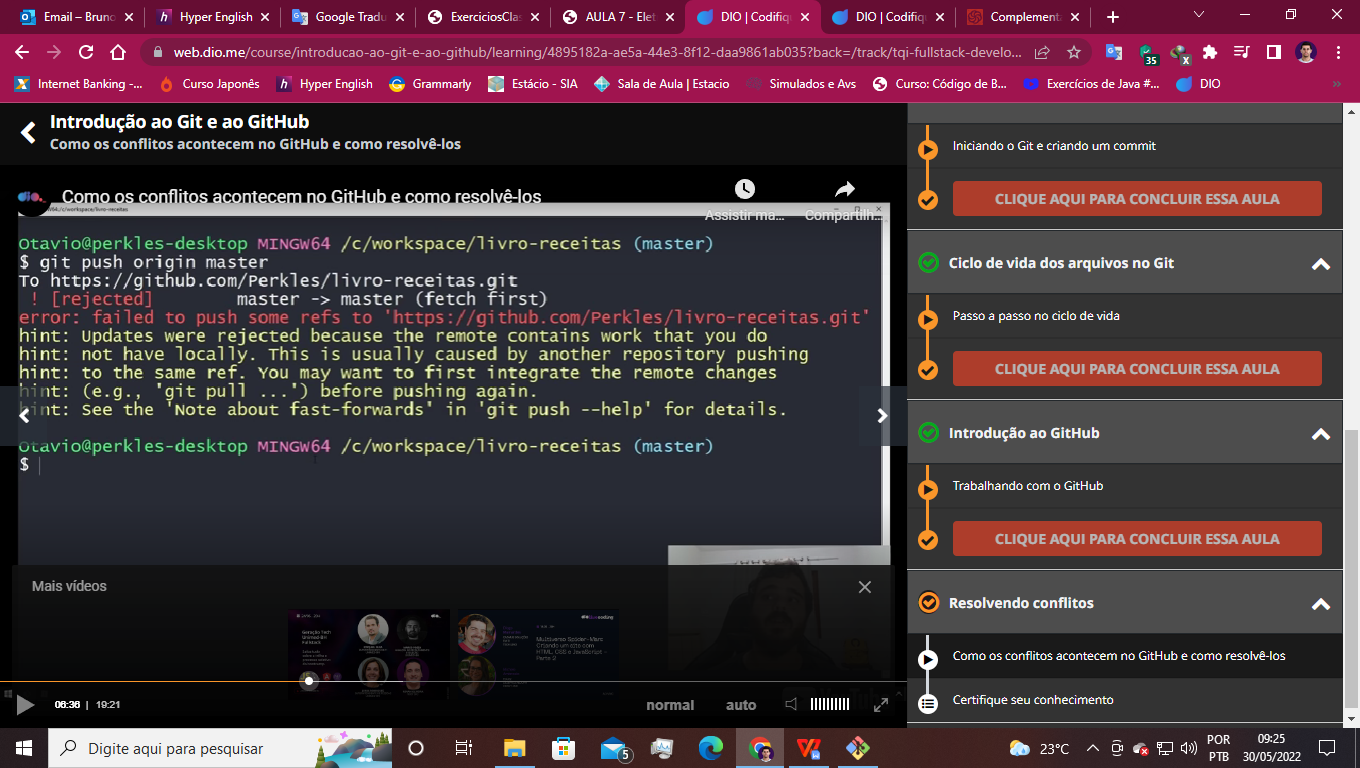
*‘Push’* de empurrar mesmo; *‘origin’* que é o nome do nosso repositório; e *‘master’* que é como o Git classifica o repositório.

Depois disso o Git vai dizer que não tem conhecimento do sha e pergunta se realmente queremos continuar, digitamos yes e depois ele vai pedir a chave privada que criamos para o conectar o nosso Git ao GitHub. Daí é só digita-la e o nosso repositório já estará presente no repositório remoto.

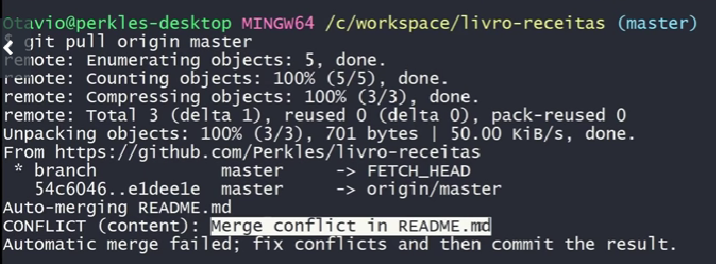
***Resolvendo Conflitos***

Quando duas ou mais pessoas estão trabalhando no mesmo arquivo contido no GitHub e ambas, em suas próprias máquinas, fazem alterações na mesma linha, quando uma delas envia as mudanças para o repositório remoto e a outra continua a editar em sua máquina aversão mais antiga do código (há que os dois estavam trabalhando antes), então quando esta última termina as edições e envia para o GitHub, o GitHub vai mandar uma mensagem informando que existe uma versão mais nova e que será necessário baixá-la e fazer as mudanças manualmente.

O erro em questão:



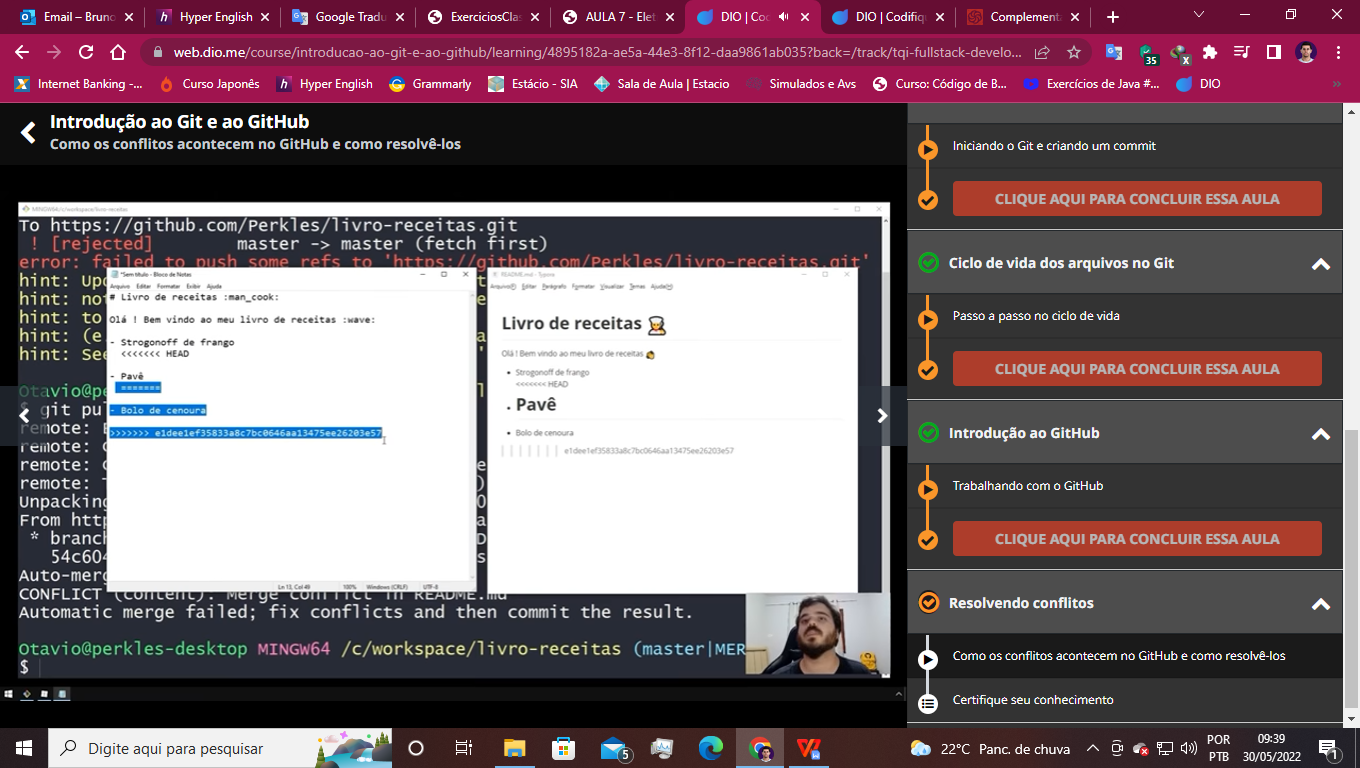
Para resolver isso fazemos o que é indicado na própria mensagem acima, damos o comando ***git pull origin master***, dessa forma o Git vai trazer as atualizações que já estão no repositório remoto e soma-las as alterações feitas por você. No entanto, se houver alterações na mesma linha, será necessário fazer as mudanças manualmente, como diz a mensagem abaixo:



Nesse caso, quando abrirmos o arquivo na nossa máquina ele irá apresentar os seguintes caracteres:

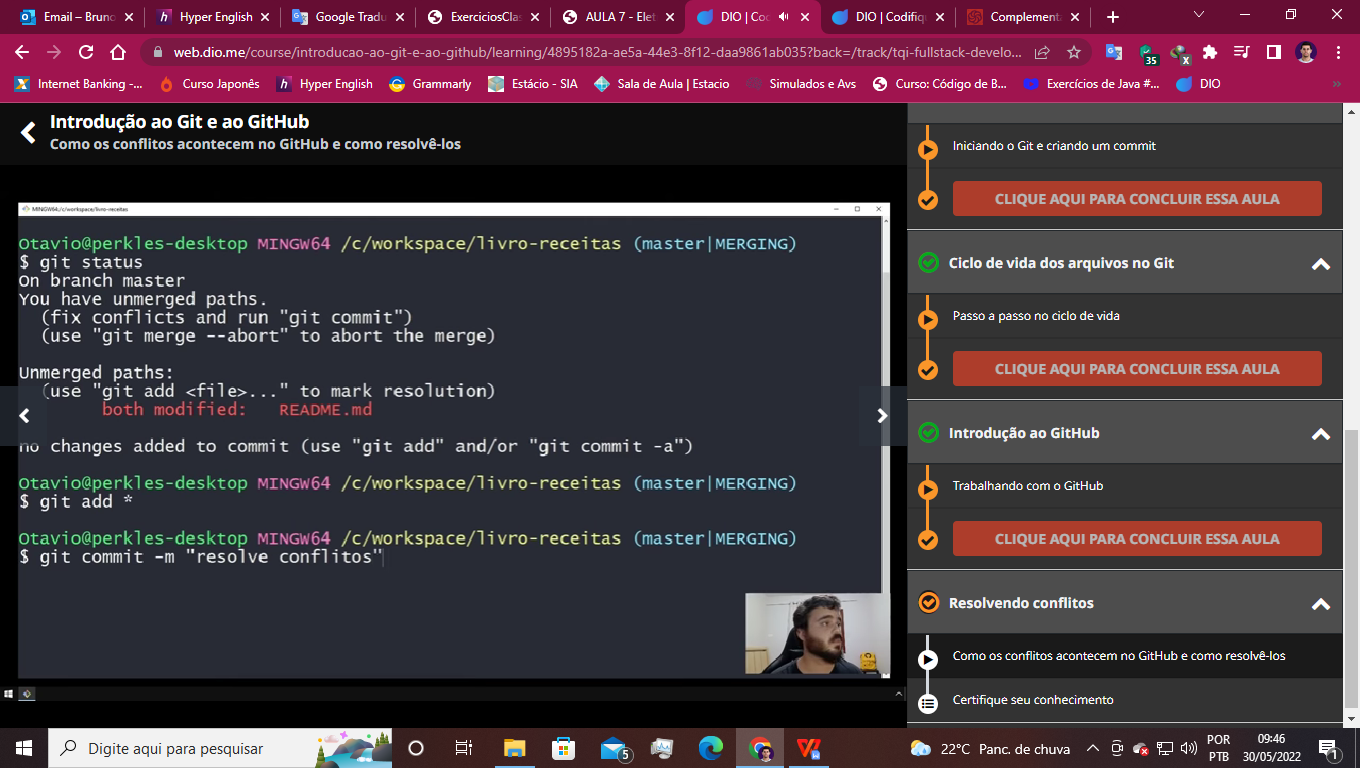
* ***<<<<<<< HEAD*** - Abaixo dele estão as nossas modificações.
* ***======*** ou ***\_\_\_\_\_\_\_\_*** - Abaixo destes estão as modificações do outro colaborador.

Então nós decidiremos quais alterações fazer, se uma, ou outra, ou ambas.



Depois de feitas as alterações o git vai nos apresentar ainda um conflito, dizendo que *both modified* (ambos modificados), porque haverá uma versão no GitHub e outra na nossa máquina. Para resolver isso basta commitar.

Assim:



Feito isso basta empurrar para o GitHub nossa versão e ele estará atualizado:

***git push origin master***

***OUTROS COMANDOS ÚTEIS***

**Clonar Repositório** *- git clone urlDoRepositórioAClonar.com*

Para verificar através do GitBash se houver alteração no arquivo no servidor remoto podemos usar o ***git fetch origin***. Ele, como o *git pull*, baixa as atualizações do arquivo presentes no servidor remoto, mas não as aplica no arquivo em que estamos trabalhando, apenas sincroniza com o banco de dados da pasta .git e dessa forma nos permite ver o número de alterações realizadas por meio do número de commits. Assim, usando o comando ***git status*** ou o ***git long*** (esse nos da mais detalhes), podemos ver a situação do nosso arquivo em relação ao arquivo remoto.