1 - Git & GitHub - O que é e para que servem

## O que é controle de versão?

Um Sistema de Controle de Versão Distribuído (**DVCS**) é o que permite um time de desenvolvedores a trabalhar em um mesmo projeto, ou em um mesmo arquivo específico, sem terem problemas de direcionamento de dados e salvamento de arquivos em versões. Ou seja, evita um amontoado de e-mails e os famosos títulos “arquivo-v.9.35.42.txt”! O **Git** é apenas um tipo de DVCS, sendo o mais comum entre os times desenvolvedores. Ele possui ferramentas para contornar a maioria dos problemas que podemos encontrar nessas situações.

O processo de entendimento do **Git e GitHub** é um processo contínuo, que continua envolvido em toda a carreira de um profissional na área de tecnologia. Precisamos entender essa tecnologia e como manipulá-la através do terminal, pois a chance de trabalharmos com ela é bem alta, e todos os projetos da trybe estarão na ferramenta.

O **Git** foi um processo criado em paralelo durante o processo de criação do próprio **Linux.** De forma simples, o Git guarda um conjunto de alterações nos arquivos através dos **commits**, que criam um histórico dessas alterações ao longo de todos os momentos em que seu **repositório** ficou ativo e isso faz com que você tenha um caminho claro, para ir e voltar no caminho do desenvolvimento desses arquivos e projetos. Dentro do Git trabalhamos em um modelo que não é baseado em cliente servidor, mas sim em **peer-to-peer**, e que o torna ágil para a gestão de projetos, pois as ferramentas para análise de **branchs (braços espelhados do projeto)**.

O **Github** não é um DVCS, mas apenas uma plataforma que hospeda a engenharia do Git. Nele encontramos as funcionalidades fundamentais para o trabalho em equipe, como quadros de Kanban, pull requests e demais ferramentas para otimizar o fluxo de desenvolvimento.

## Conceitos básicos de Git:

As **branches** são versões independentes e editaveis dos código do repositório e que funcionam como “braços” ou “divisões de caminho” da **branch master** (que é, geralmente, o **cerne de nosso projeto**). A branch master também é editável, mas precisamos ter muito cuidado com ela, sendo que na maioria das empresas, é proibido fazer alterações diretamente nela. As edições que fazemos nas branchs são chamadas de **commit** e elas são etapas das alterações dentro do código que está contido na branch.

Podemos ramificar as **branchs** em nossos projetos quantas vezes julgamos necessário e todas essas **branchs secundárias** terão seus próprios **commits**, sendo uma linha paralela do que acontece no master do projeto.

Assim que necessário, podemos agregar as branchs **secundárias,** em nossa branch **master** através do **merge,** que corresponde a fundir os dois braços paralelos do projeto em um só, respeitando as modificações de código entre uma e outra.

## Autenticações:

Os processos de autenticação são de extrema importancia para que não ocorram interferências externas, vazamento de dados ou corrompimento de projetos ao longo de nossa vida profissional. A autenticação no **GitHub** permitirá que você conecte seu terminal à plataforma e a atualize por lá, sem interferências. Assim que o processo é realizado, podemos utilizar nossas credenciais para informar ao **sistema remoto** comandos **git** dentro de nossos terminaise, ao mesmo tempo, comprovar para a plataforma que é você mesmo quem os digita.

Existem duas formas de autenticação pelo **GitHub**:

* **SSH** ou **SecureSHell**: é um protocolo de criptografia de rede que serve para transferir os dados de forma segura, mesmo em redes inseguras. Usando o protocolo SSH, você pode se conectar ao GitHub sem precisar digitar seu nome e chave de acesso pessoal a cada comando executado no terminal.
* **HTTPS** ou **Hypertext Transfer Protocol Secure**: é uma extensão do protocolo de internet HTTP que utiliza certificados digitais para autenticar os dados e permitir que sejam criptografados de forma segura.

Aqui na Trybe utilizaremos o protocolo SSH para a autenticação de dados de projetos pelo Git e GitHub.

## Do **git init** ao **git push:**

Após criar o diretório que desejamos enviar até a plataforma do GitHub, utilizamos o comando de **git init**. O comando vai criar uma pasta **.git** dentro do diretório, iniciando o direcionamento de informações para o repositório remoto. Para verificarmos se o comando **git** realmente iniciou o repositório, utilizamos o comando **git status**, que retornar algo parecido com isso (quando esta tudo certo e sincronizado):

No ramo master

No commits yet

nada para enviar (crie/copie arquivos e use "git add" para registrar)

Ao criarmos um arquivo dentro do diretório direcionado para o **git,** só conseguimos adicioná-los através do comando **git add**, no qual “**git add .”** adiciona todos os arquivos que modificamos do parâmetro base ou “**git add meu\_arquivo.txt”** (onde ‘meu\_arquivo.txt’ é o nome do arquivo que você quer direcionar) adiciona somente o arquivo em específico. Após adicionar a modificação, é indicado abrir um “mark point” através de um **commit**! O comando **git commit -m “mensagem que você quiser”** cria essa espécie de checkpoint na branch que estamos trabalhando, com a mensagem personalizada que achamos melhor, para indicar o que alteramos no trabalho.

Após criar o novo repositório dentro da plataforma do GitHub, precisamos conectá-lo à nossa máquina local e ele vem através do comando **git remote add origin** [**git@github.com**](mailto:git@github.com)**:user-github/repo-name.git** , sendo o **origin** um apelido estabelecido por convenção para o repositório (no qual podemos apelida-lo conforme nossas necessidades) e o link com email - somado ao user.name - é a URL gerada dentro do site do GitHub indicado ao criar o repositório e deve ser copiado de lá.

Para verificar se tudo aconteceu normalmente, utilizamos o **git remote -v**, assim o terminal retorna os diretórios de origem (maquina) e remoto (GitHub).

Assim os repositórios estão criados e conectados, mas precisamos enviar alterações feitas em nossas máquinas para o terminal remoto. É importante saber sempre, através do comando **git status** se o decorrer do projeto está sempre sincronizado. Caso esteja tudo adicionado e comitado dentro dos arquivos, é hora de passar a informação para o repositório. Essa passagem é feita pelo comando **git push origin master**, que empurra para o repositório remoto (origin) a atualização na branch (no caso desse comando, a branch **master**).

2 - Git & GitHub - Entendendo os Comandos:

## Baixando Repositórios com o **git clone:**

Entender o conceito de **git** e saber seus comandos será de extrema importância ao longo do curso da trybe e de nossas vidas profissionais, por isso a prática é de extrema importância!

Se quisermos copiar um repositório **git** existente no **github,** como um projeto que estamos trabalhando ou quisermos contribuir, é necessário dominarmos o comando **git clone**. Mais do que baixar o repositório desejado, o git recebe cópias de todos os dados que o servidor possui. Cada versão de arquivo é obtida quando rodamos o comando.

Ao comandarmos **git clone $URLDESEJADA** iremos receber um clone exato do projeto que queremos participar e devemos manejá-lo de forma ética, **nunca fazendo alterações na master**.

## Do **git log** ao **pull request:**

Aqui vamos aprender a lidar melhor com os **commits, adds e pushes** que aprendemos anteriormente, para aprimorar nossa prática.

### /git log

O  **git log** é o comando que permite visualizarmos se existem ou não **commits** dentro da branch. É o comando que permite se existem “safe points” dentro da brach que estamos trabalhando e mostra todo o histórico de atualização de commits dentro do que estamos fazendo. O comando mostra o número identificador do commit, identifica quem criou, mostra a data da atualização e permite a visualização das atualizações criadas dentro da branch.

### /git remove

O **git remove** é o comando para removermos, apagarmos e ressuscitarmos os arquivos que desejamos esquecer (ou renovar) em nossos **commits**. Para apagar arquivos simples, utilizamos o comando **git rm $NOME.DO.ARQUIVO** para apagarmos ele de nosso próximo commit (e aparece ao utilizar o comando git status como ‘algo removido’). Se você apagou algum arquivo e se arrependeu - ou simplesmente quer voltar atrás -, existe o comando **git log --diff-filter=D --summary** (**essa flag filtra\* os arquivos diferentes\* existentes entre diferentes commits que contenha arquivos iguais\* ao status de Deletado\* e que sejam exibidos em forma de Sumário\***), assim o **git log** mostra pra gente os commits que tenham arquivos que foram deletados. Assim, pegamos os quatro primeiros caracteres do “nome/IDNumber” de nosso commit e executamos o comando **git checkout $NOME/IDENUMBER~1 $NOME.ARQUIVORESSUCITADO** para passarmos o commit ao status de “ok” (**o ~1 é o identificador para validação do comando. Se ~0 ele continua invalidado**). Se inspecionarmos o diretório vemos que o arquivo voltou a existir dentro dele e podemos seguir os trabalhos a partir daí, porém não adicionado e commitado. Ao visualizarmos o **git log** notamos que a nossa recuperação também é identificada no nosso histórico.

### /git ignore

Utilizamos o comando **git ignore** para ignorarmos arquivos que não queremos ou necessitamos dentro de nossas pastas de projetos. Pode ser por serem apenas de visualização, arquivos temporários ou arquivos muito pesados para serem passados a todos os membros do grupo de trabalho. Antes de tudo, precisamos criar o arquivo oculto **.gitignore** para começarmos essa função. Após criarmos, utilizamos **cat >> .gitignore** para editar o arquivo oculto e adicionamos a ele os nomes, de diretórios ou \*extensões de arquivos que queremos ignorar. Ao realizarmos o comando ignore, podemos adicionar os arquivos que realmente queremos subir aos nossos commits. Não podemos esquecer de commitar o arquivo oculto do **.gitignore** senão o sistema de versionamento não reconhece a função. Se desejarmos ignorar mais algum arquivo após o commit, precisamos apenas editar o arquivo novamente e adicioná-lo ao commit seguinte.

Existe um site chamado [www.toptal.com/developers/gitignore](http://www.toptal.com/developers/gitignore) para automatizar a tarefa do **.gitignore**. Nele, adicionamos qual é a ferramenta que utilizamos e ele cria sozinho a edição do arquivo oculto .gitignore, cabendo a nós apenas editar o arquivo e adicionar essa edição dentro dele!

### /git push | /git pull | /git fetch

Ao criarmos um novo repositório dentro do GitHub, geralmente utilizamos o comando **git push -u $ORIGIN $BRANCHMASTER** para empurrarmos todo o conteúdo da nova branch master ao repositório remoto. O GitHub só atualiza em sua página um diretório que contém arquivos ativos. Com o **git clone + chave SSH do projeto** podemos clonar esse repositório em nossas máquinas ou usuários a parte.

Para “empurrarmos” os **commits** que estamos trabalhando dentro do projeto para o repositório remoto, utilizamos o **git push (sem o origin$ master$ pois ele foi feito inicialmente)** e assim atualizamos a branch que estamos trabalhando. Se outro usuário estiver trabalhando no mesmo trabalho e deseja saber se existem atualizações no commit em que está atualmente, usamos o comando **git fetch** que verifica se existe alguma modificação realizada no repositório remoto e quantos commits diferentes existem. O **git pull** também verifica se existe alguma atualização, mas “puxa” todo o conteúdo do github, fazendo uma espécie de **merge** com sua branch atual, automaticamente.

### Pull Request no GiitHub

Dentro de grandes projetos não podemos simplesmente atualizar os códigos de branchs que estamos trabalhando sem que eles sejam verificados. Para fins de boas práticas, o GitHub conta com uma ferramenta para solicitar o **pull** de arquivos dentro das **branchs** para que o usuário dono do repositório remoto consiga conferir e dar o aval para essa atualização.   
 Conseguimos fazê-lo dentro da própria plataforma do GitHub sendo ele um botão, dentro da página principal do repositório, que fica dentro do menu “branch”. Dentro dessa função, utilizamos o menu suspenso para indicar o **merge** que queremos fazer, damos um título de commit a ela e podemos descrevê-la. Esse processo depende de aprovação e, até ser aprovado, conseguimos comentá-la para que outras pessoas que estão trabalhando nessa branch consigam visualizar, ajudar ou refletir sobre essa modificação.

3 - Internet

Antes de estudarmos sobre programação Web, precisamos aprender mais sobre o meio em que estamos e como a Internet conhece, dando um conceito geral sobre sistemas e ecossistema nela disponível. Como a maioria das aplicações para web é desenvolvida no modelo *cliente & servidor* , temos que saber como funcionam as trocas de dados para sermos mais efetivos quando colocarmos um projeto ‘no ar’ ou ‘em produção’.

## As peças que montam a Internet:

A partir do momento em que escrevemos o nome de um site no nosso navegador, existe um processo longo de validação de protocolos e um fluxo de informação de pacotes, e os principais termos estão abaixo:

### Cliente:

Firefox, Mozilla, Chrome (nunca o edge)... Seja qual navegador você utilize para digitar o site, ele é o **cliente**. O cliente é a aplicação que está conectada à Internet. A função principal do cliente é traduzir e renderizar os pacotes enviados até ele e traduzi-la para outro computador denominado Servidor Web. Você pode pensar no seu computador como o **Cliente** dentro do modelo ***Servidor - Cliente*** . Todo computador tem seu único identificador e ele se chama **IP.**

### Servidor:

O **Servidor** é um "super computador" conectado à Internet. É onde nosso navegador obtém o requerimento ao enviar arquivos e ele também possui seu próprio IP. O Servidor espera por requisições de outras máquinas (Cliente) mas, diferente de nossos computadores domésticos, um Servidor tem um software específico em execução que irá dizer como ele responderá a uma Requisição do Cliente. A principal função do Servidor é **armazenar, processar e entregar** pacotes de dados aos Clientes.

### Endereço IP:

A sigla **IP** refere-se à Internet **Protocol**. O endereço IP é um identificador singular que existe em todo dispositivo conectado à Internet e está conectado na rede **TCP/IP**. O IP é utilizado para localizar e identificar um usuário, conectando você com outros dispositivos. O IP possui um dígito de quatro campos (ex: 244.155.65.2) e são chamados de **endereço lógico,** podendo ser localizado e transformado em endereço físico através de um software. O endereço físico faz parte do seu hardware e é conhecido por **Endereço de Controle de Acesso à Mídia (MAC address).**

### TCP/IP:

É sigla para **Transmission Control Protocol / Internet Protocol.** O conjunto de regras (protocolo) é utilizado para transmitir dados de rede e é o protocolo mais utilizado no mundo web.

### ISP:

O **Internet Service Provider** é o nosso provedor de Internet. É o nome das empresas que fornecem acesso à Internet, como Claro/Net, Oi/Velox e etc.

### DNS:

**Domain Name System** é um dispositivo com uma base de dados distribuída e gerencia os nomes de serviços, computadores ou qualquer dispositivo conectado à internet. Ele relaciona **o endereço nominal** com o **número de IP.** Os servidores DNS são os responsáveis por localizar e traduzir números às buscas por sites, quando os digitamos no navegador.

### Port Number:

É um número de **16 bits** utilizado para identificar uma porta específica em um servidor e está sempre associada a um número de IP.

### Host:

/o **Host** é qualquer computador conectado à rede, seja como cliente, seja como servidor ou qualquer outro gadget. Cada host tem um IP único. É comum **confundir host com servidor, mas eles são coisas diferentes!**

### HTTP:

**Hyper-test Transfer Protocol** é o protocolo utilizado para a comunicação entre navegadores e servidores na internet.

### URL:

É o **Uniform Resource Locator** e identifica um recurso web específico (como um site). Como no exemplo https://www.google.com/travel/flights , o URL identifica o protocolo que será utilizado para comunicarmos com o servidor (https), o nome do host (google.com) e o recurso que queremos acessar (travel/flights).

Podemos descrever o processo de protocolos ao digitarmos google.com no nosso navegador assim:

