Faculdade Metropolitana da Grande Fortaleza – Fametro –

Fametro
Formar para Transformar

Desenvolvimento de Aplicações Corporativas

João Júnior

Fametro
Formar para Transformar

Unidade 2 – Git

Git

Controle de Versão

- O que é Controle de Versão?
- Em inglês: Version Control Systems (VCS)
- É um sistema que registra alterações em um arquivo ou conjunto de arquivos ao longo do tempo para que você possa lembrar de versões específicas mais tarde.

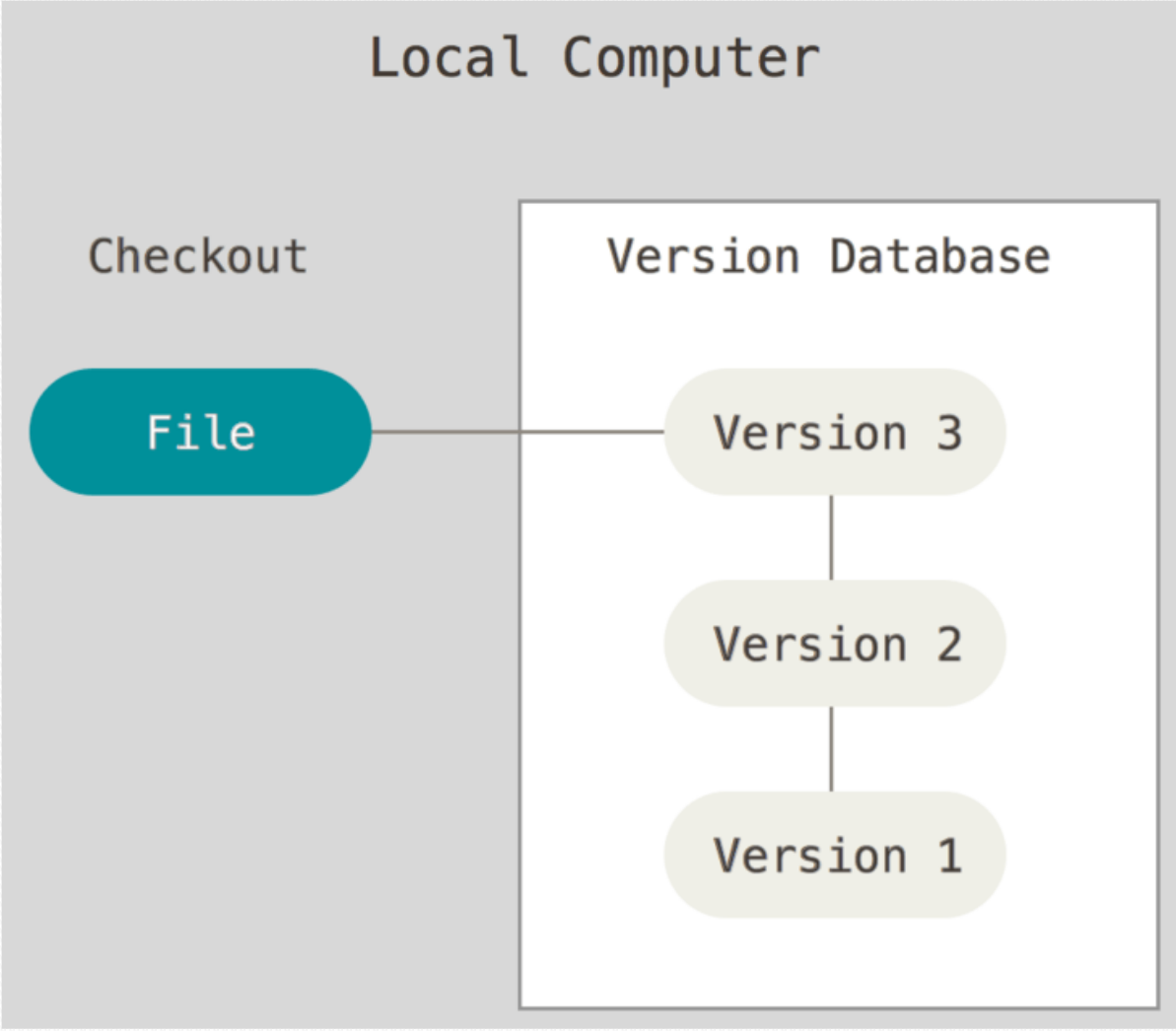
Git

Controle Local

- Muitas pessoas usam um método de controle de versão onde simplesmente copiam arquivos em outro diretório (talvez um diretório com controle de tempo, se forem espertos).
- Esta abordagem é muito comum porque é simples. Entretanto, também é incrivelmente propensa a erros.
  - Ex.: É fácil esquecer em qual diretório você está dentro e acidentalmente gravar no arquivo errado ou até mesmo copiar arquivos que não gostaria.
- Para lidarem com essa situação, os desenvolvedores há bastante tempo construíram VCSs que tinham um simples banco de dados que mantinham todas as mudanças nos arquivos em um controle de revisão, conforme a próxima imagem:

Git

Controle Local



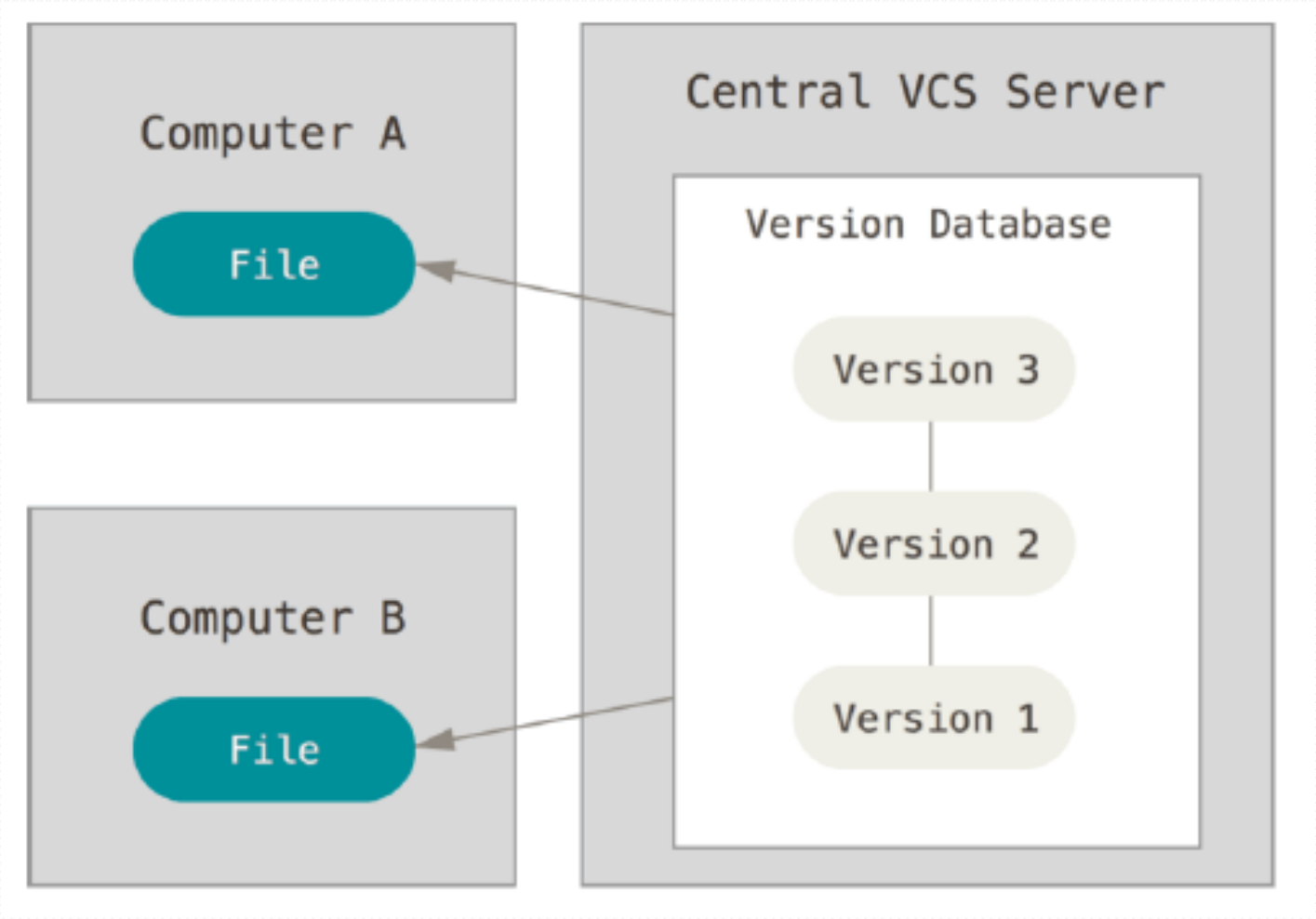
Git

Sistemas de Controle de Versão Centralizados

- O próximo desafio encontrado era a necessidade de colaborar com desenvolvedores em outros sistemas.
- Para lidar com este problema, foram desenvolvidos Sistemas de Controle de Versão Centralizada (Centralized Version Control Systems - CVCSs)
  - Exemplos: CVS, Subversion e Perforce,
- Todos eles possuem um único servidor que contém todos os arquivos versionados e um número de clientes que fazem *check-out* de arquivos a partir desse lugar central.
- Por muitos anos, este tem sido o padrão para controle de versão.

Git

Sistemas de Controle de Versão Centralizados



Git

Fametro
Formar para Transformar

Sistemas de Controle de Versão Centralizados

- Esta configuração oferece muitas vantagens, especialmente sobre VCSs locais.
- Exemplo: todo mundo sabe até certo ponto o que todos estão fazendo no projeto.
- Os administradores têm controle refinado sobre quem pode fazer o quê e é muito mais fácil para administrar um CVCS do que lidar com bancos de dados locais sobre cada cliente.

Git

Fametro
Formar para Transformar

Sistemas de Controle de Versão Centralizados

- Entretanto, a desvantagem mais óbvia é o ponto único de falha que o servidor centralizado representa.
- Se esse servidor cai por uma hora, durante esse tempo ninguém pode colaborar com nada ou até salvar as alterações de versão para qualquer coisa que eles estejam trabalhando.
- Se o disco rígido central banco de dados está em corrompido e backups apropriados não foram mantidos, perdemos absolutamente tudo, toda a história do projeto, exceto o que individualmente os integrantes do projeto mantiveram em suas máquinas locais.
- Sistemas de VCS locais sofrem deste mesmo problema: sempre que você tem toda a história do projeto em um único lugar, você corre o risco de perder tudo.

Git

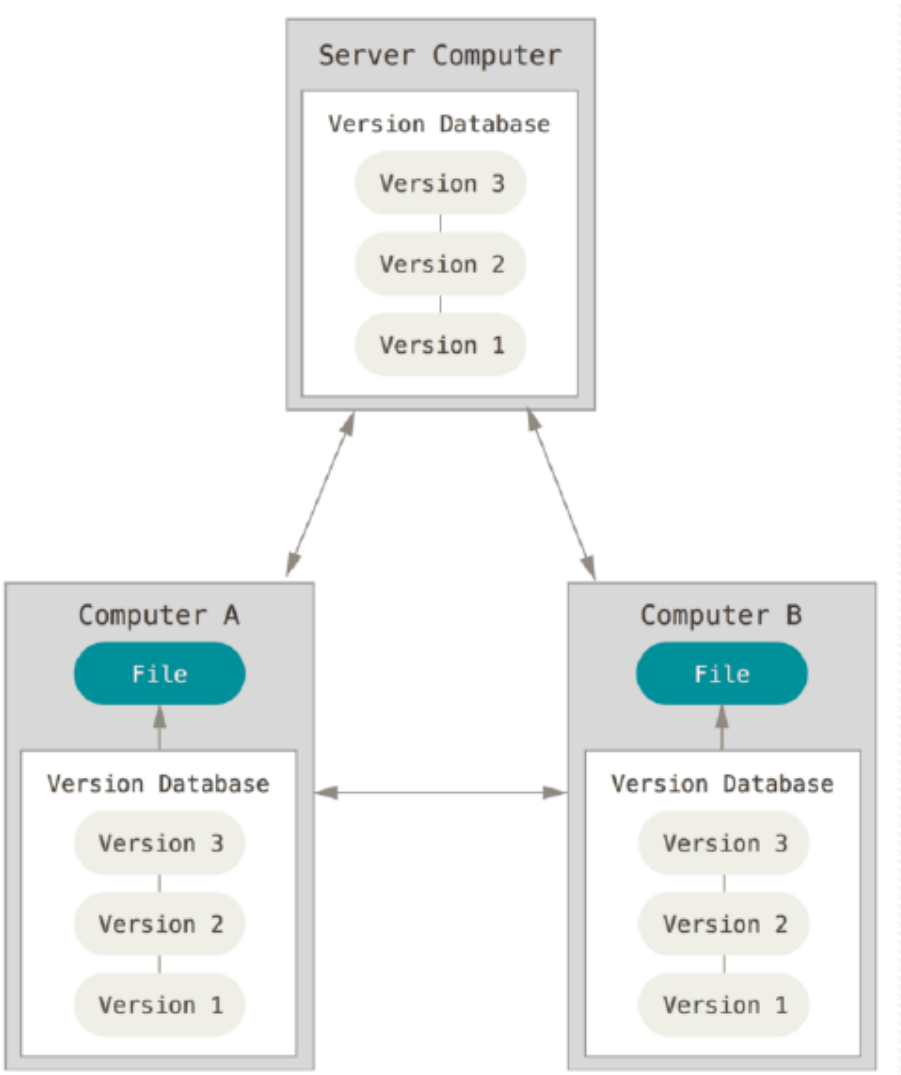
Fametro
Formar para Transformar

Sistemas de Controle de Versão Distribuídos

- Aqui é onde os Sistemas de Controle de Versão Distribuídos entram - Distributed Version Control Systems (DVCSs).
- Exemplos: Git, Mercurial, Bazaar e Darcs.
- Os clientes não apenas fazem o *check out* da última versão dos arquivos. Eles espelham totalmente o repositório!
- Assim, se qualquer servidor morrer, e esses clientes estavam colaborando por meio dele, qualquer um dos repositórios de cliente pode ser copiado de volta para o servidor para restaurá-lo.
- Cada clone é realmente um **backup** completo de todos os dados.

Git

Sistemas de Controle de Versão Distribuídos



Git

Sistemas de Controle de Versão Distribuídos

- Além disso, muitos desses sistemas lidam muito bem em terem vários repositórios remotos com os quais possam trabalhar e colaborar com diferentes grupos de pessoas de maneiras diferentes ao mesmo tempo dentro do mesmo projeto.
- Isso permite que você configure vários tipos de fluxos de trabalho que não são possíveis em Sistemas Centralizados, tais como os modelos hierárquicos.

Git

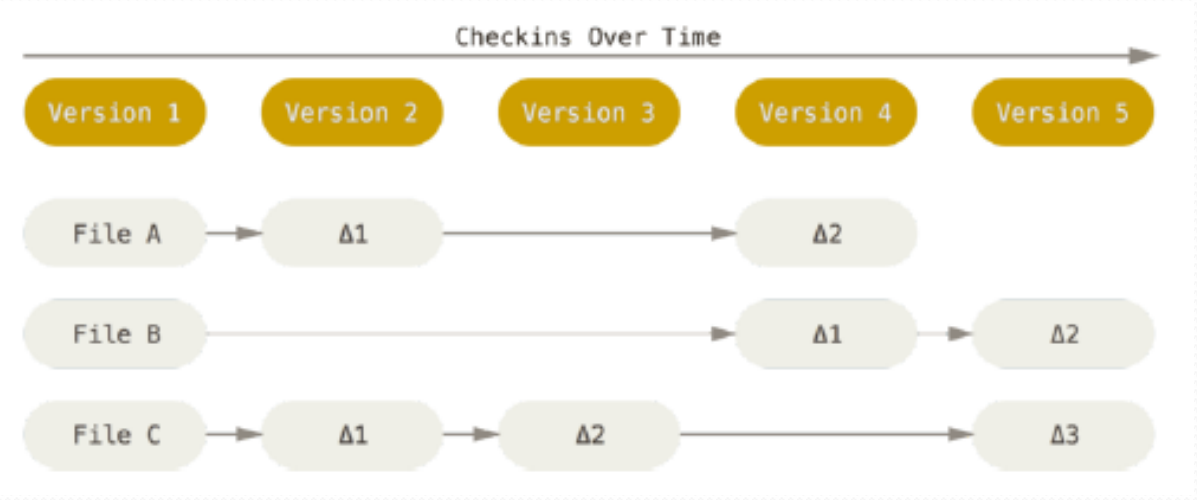
O que é o Git?

- Antes de tudo: limpe sua mente enquanto aprende a usar o Git.
- Mesmo que em um primeiro momento a interface possa parecer a mesma dos VCSs, é importante entender os conceitos do Git para não se confundir.
- Primeira coisa: **snapshots** e não **diferenças**
  - A maior diferença entre o Git e os demais VCSs é a forma como o Git entende o que significa um dado.
  - Conceitualmente, a maioria dos sistemas gravam informação como uma "lista de arquivos baseados em mudanças".
  - O Git entende que os dados são como um conjunto de snapshots de um sistema de arquivos em miniatura.
  - Vamos comparar as duas próximas imagens.

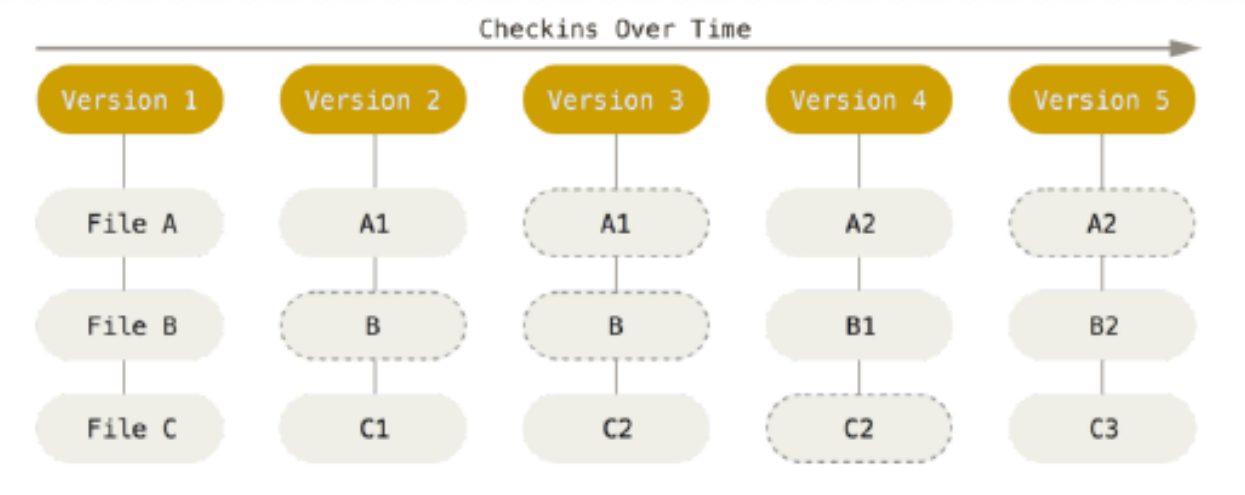
Git

O que é o Git?

VCSs:



Git:



Git

Quase todas as operações são locais

- A maioria das operações do Git precisam somente dos arquivos e recursos locais para realizar ação.
- Geralmente nenhuma informação é necessária de outro computador ou da rede.
- Para quem sofria de conexão quando utilizava o SVN já deve estar dando graças a Deus por no Git não existir isso! ☺

Git

Fametro
Formar para Transformar

Integridade sempre!

- No Git tudo é feito o checksum* antes de ser gravado e é então referenciado para aquele checksum feito.
- Isso significa que é impossível mudar o conteúdo de qualquer arquivo ou diretório sem o Git tomar conhecimento a respeito.
- O mecanismo de checksum usado é o SHA-1 hash que é uma string de 40 caracteres composto por hexadecimais, por exemplo: `24b9da6552252987aa493b52f8696cd6d3b00373`

**Checksum é um código usado para verificar a integridade de dados transmitidos através de um canal que possa ter ruídos ou então armazenados em algum meio por algum tempo.*

Git

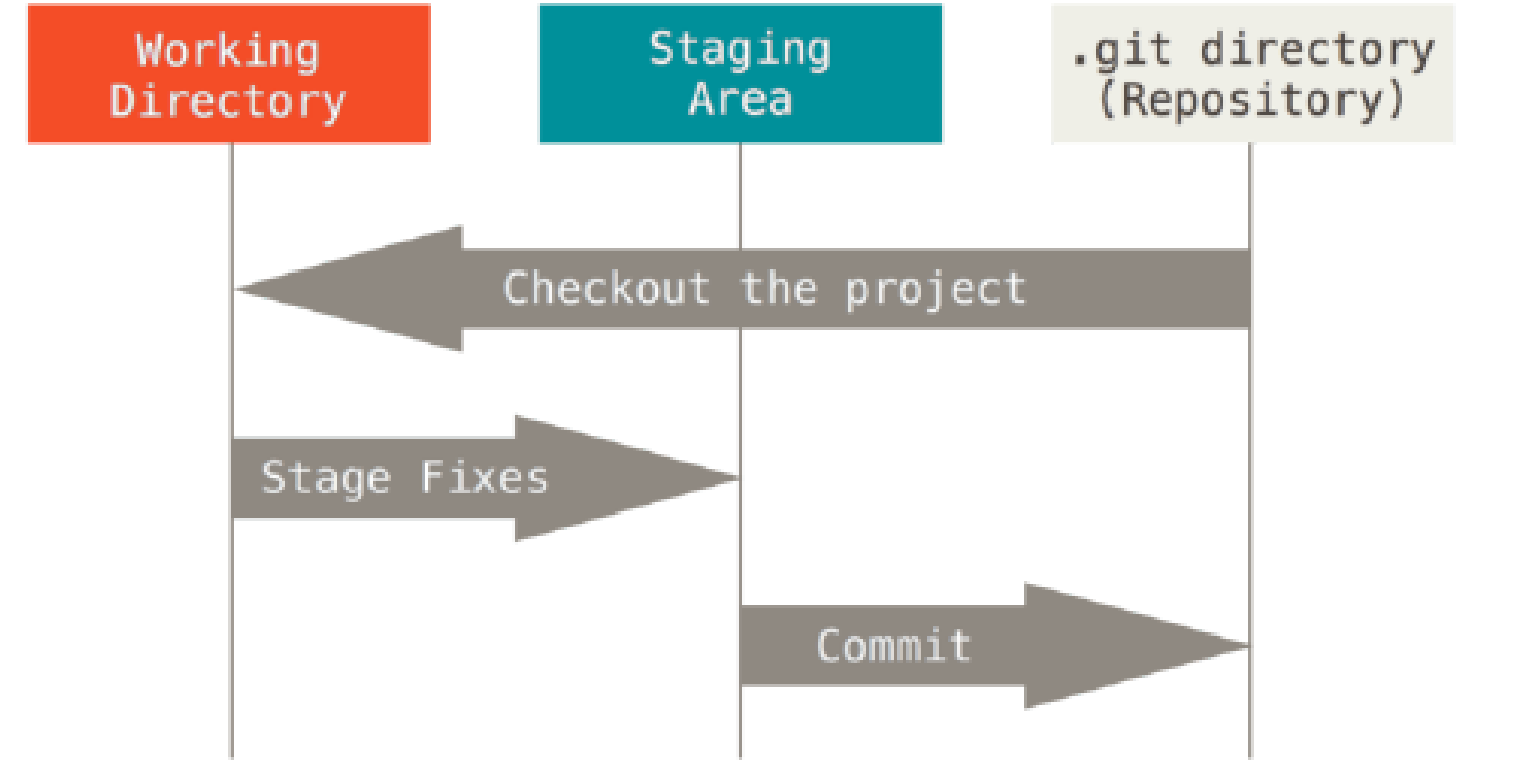
Fametro
Formar para Transformar

Os três estados

- Atenção para a coisa mais importante do Git!
- O Git possui três principais estados em que seus arquivos podem ficar:
  - **Committed** = está guardado na base de dados local.
  - **Modified** = você mudou o arquivo, mas não commitou ainda para a base dados.
  - **Staged** = você marcou um arquivo modificado na sua versão atual para ser levado no próximo commit do snapshot.
- Isso nos leva para as três seções principais do Git:
  - Git Directory (diretório Git)
  - Working Directory (diretório de trabalho)
  - Staging area (área de teste)

Git

Os três estados



Git

Os três estados

- Git Directory
  - É onde o Git grava os metadados e objetos da base de dados para o seu projeto.
  - Essa é a parte mais importante do Git.
  - É o que é copiado quando você clona um repositório de outro computador.
- Working Directory
  - É um checkout único de uma versão do projeto.
  - Esses arquivos são puxados da base de dados do Git Directory e colocados no disco para você usar ou modificar.
- Staging Area
  - É um arquivo geralmente contido no seu Git Directory que grava as informações sobre que irá no seu próximo commit.

Git

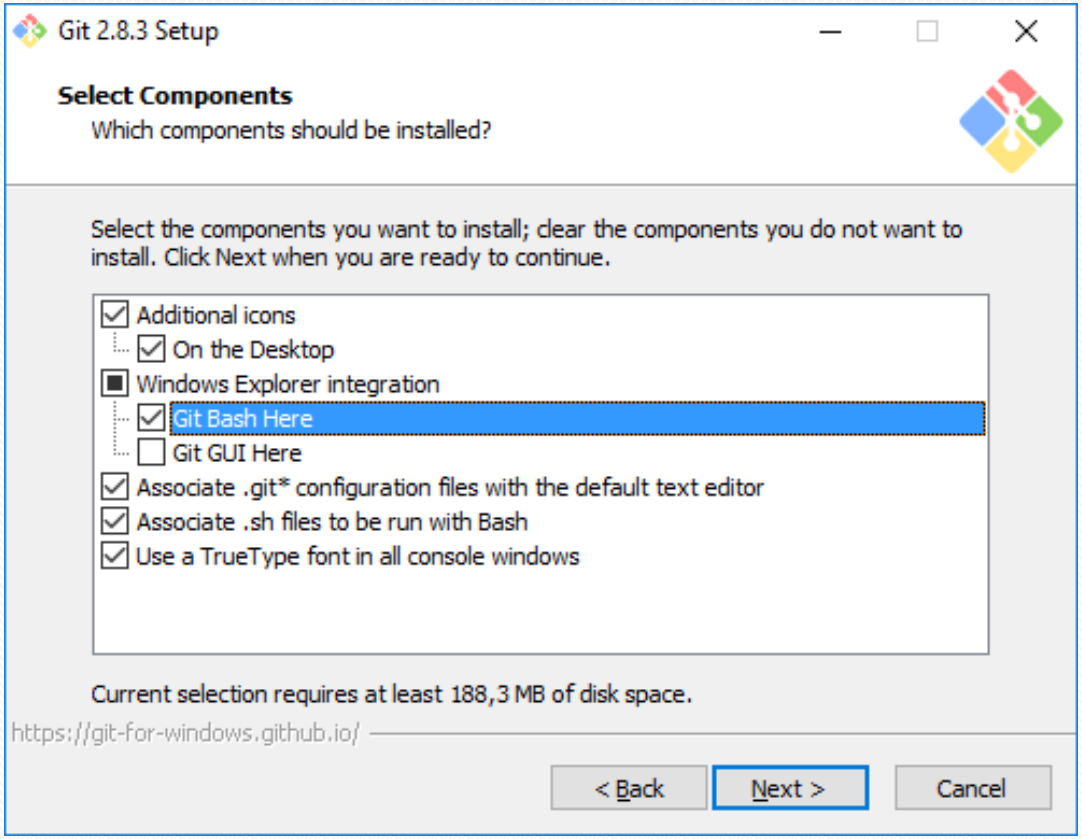
Os três estados

- O workflow do Git basicamente é:
  1. Você modifica os arquivos no seu diretório de trabalho;
  2. Você *stage* (test) os arquivos adicionando snapshots deles para a sua Staging Area
  3. Você faz o *commit* que pegas os arquivos como eles estão na Staging Area e grava o snapshot permanentemente no seu Git Directory
- Se uma versão em particular de um arquivo está no Git Directory, ela é considerada **commitada**.
- Se ela foi modificada e adicionada na Stage Area, ela está **staged**.
- Se ela foi alterada desde do checkout, mas não foi staged, ela está **modificada**.

Git

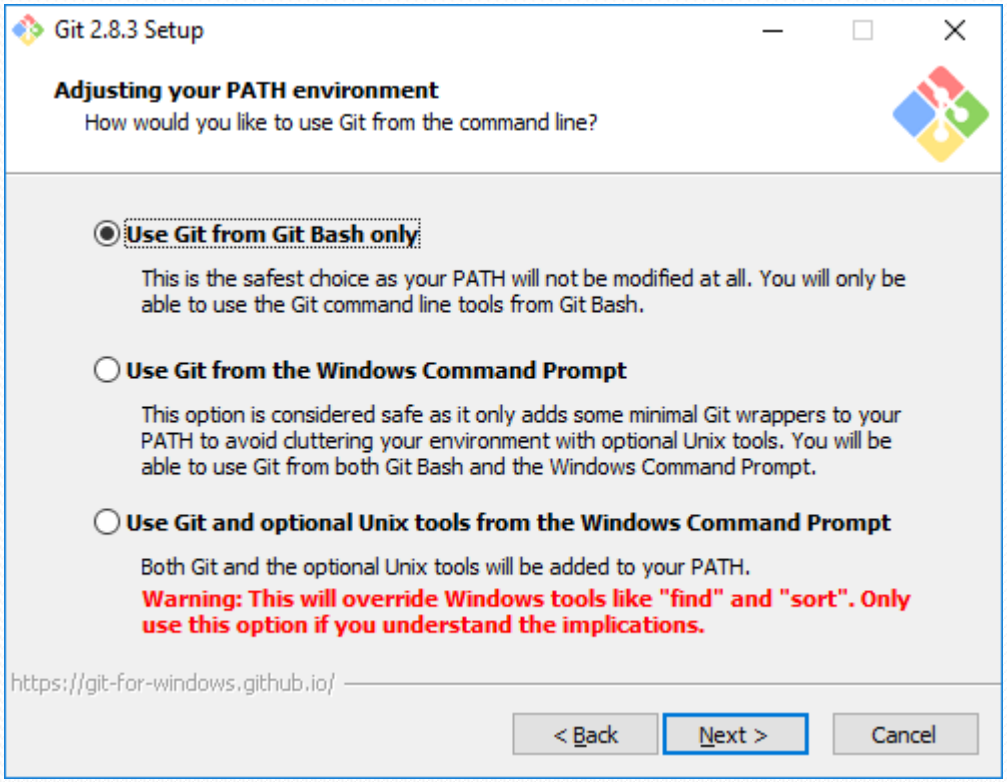
Instalando o Git - Windows

- A versão mais recente fica em http://git-scm.com/download/win



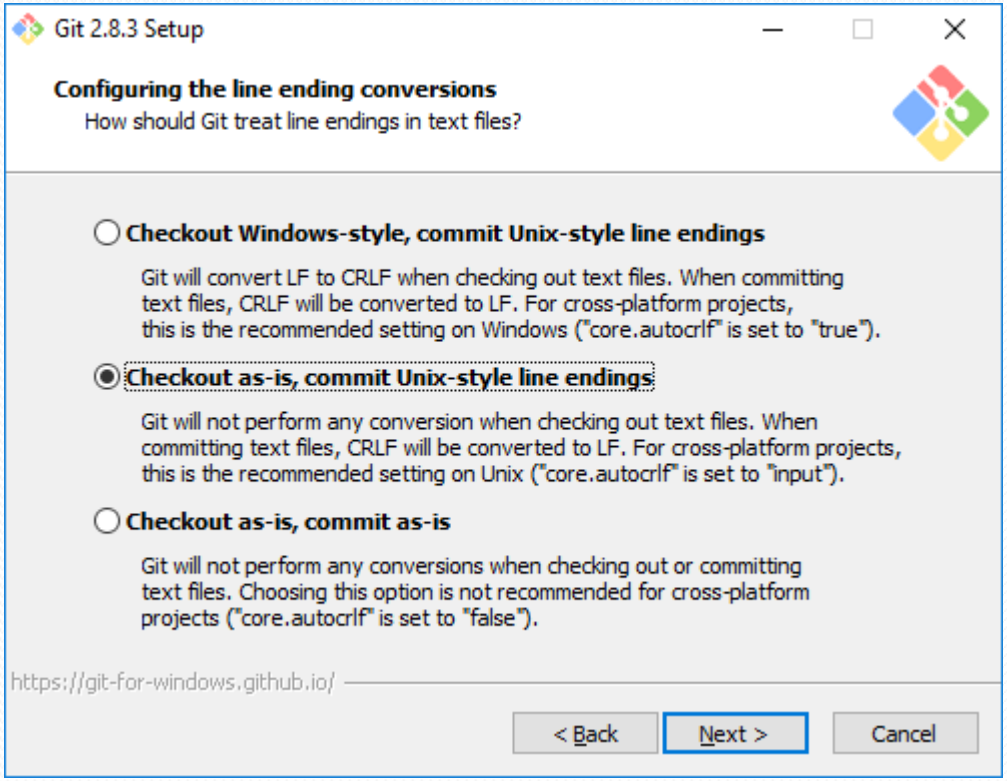
Git

Instalando o Git - Windows



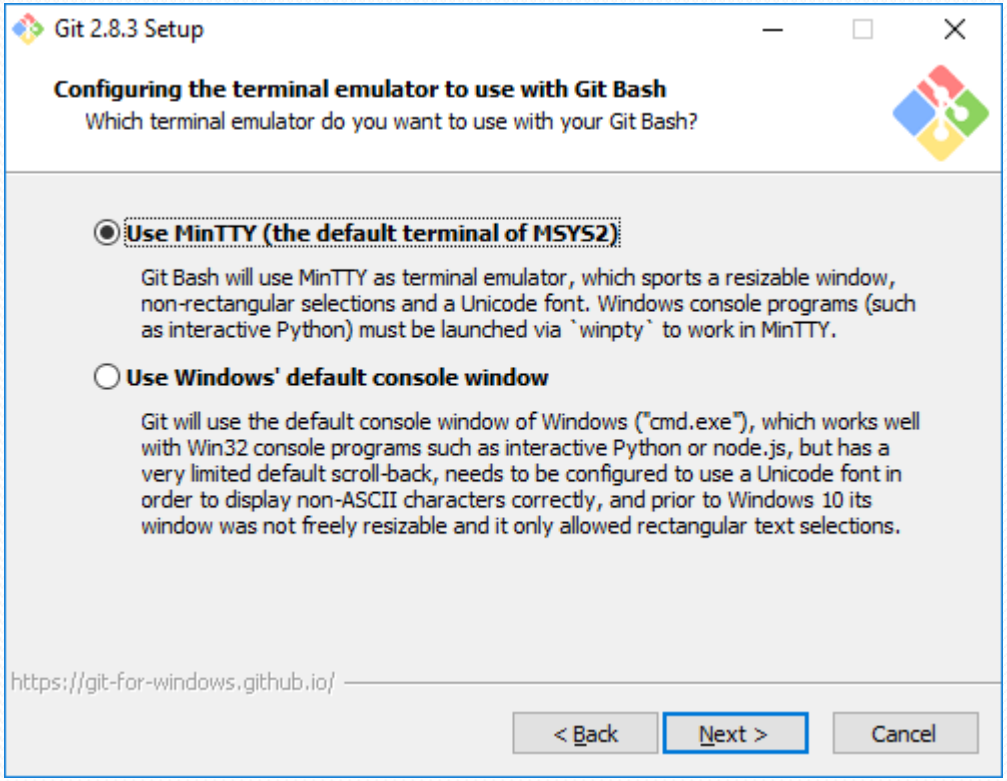
Git

Instalando o Git - Windows



Git

Instalando o Git - Windows



Git

Criando um repositório

- Agora que temos o Git instalado, vamos criar um repositório.
- Crie uma pasta, clique com o botão direito em cima dela e depois clique em "Git Bash Here"
- Precisaremos identificar os nossos commits. Para isso, no digite no console aberto:

```
git config --global user.name "Nome Sobrenome"
git config --global user.email "email@provedor.com"
```

- Agora vamos iniciar o repositório para essa pasta que acabamos de criar:

```
git init
```

Git

Mudando o status do repositório

- Vamos ver como está o status do repositório:

```
git status
```

- Aparecerá uma mensagem informando que não há nada para commitar, pois o **Working directory** está limpo.
- Agora, vá no Windows Explorer e crie um arquivo "exemplo.txt" dentro da pasta que você criou.
- Veja novamente como está o status do repositório:

```
git status
```

- Perceba que a mensagem agora é outra!
- O Git informa que existe um arquivo **untracked**. Ou seja, é um arquivo que o Git desconhece o seu histórico.

Git

Mudando o status do repositório

- Então, vamos adicionar esse arquivo no repositório:

```
git add exemplo.txt
```

- Vamos olhar como está o status agora:

```
git status
```

- Perceba que novamente a mensagem mudou!
- Agora o Git fala que existem mudanças a serem commitadas. Pra isso, digite:

```
git commit -m “Meu primeiro commit!"
```

- Em seguida o Git informa mensagem de sucesso.

Git

Conectando a um servidor remoto

- Se desejamos colocar nossos repositórios em um servidor que não tem contas para todos da nossa equipe para quem quiser ter acesso de escrita, então devemos configurar o acesso SSH para eles.
- Existem algumas maneiras que você pode dar acesso a todos em sua equipe.
- A primeira é a criação de contas para todos, que é uma solução direta, mas pode ser complicada.
- Então, usaremos chave pública SSH.

Git

Gerando uma Chave Pública SSH

- Diversos servidores Git fazem autenticação usando chaves públicas SSH (Secure Shell).
- Para isso, cada usuário deve gerar uma chave.
- Esse processo é o mesmo independente do SO.
- Primeiro, veremos se já não temos uma chave.
- Por padrão, as chaves SSH são gravadas na pasta **~/.ssh**

```
cd ~/.ssh
ls
```

- Procure por um par de arquivos com nome **id_dsa** ou **id_rsa** e outro com o mesmo nome com a extensão **.pub**
- Esse arquivo **.pub** é a sua chave pública e o outro é a sua chave privada.

Git

Gerando uma Chave Pública SSH

- Como acabamos de instalar o Git, certamente não existem chaves SSH, logo, precisaremos criar as nossas. Pra isso, digite:

```
ssh-keygen
```

- Primeiro pergunta onde você quer salvar a chave (.ssh/id_rsa);
- Depois pede duas vezes uma senha que você não precisa informar caso não queira para usar a chave depois;
- Agora cada usuário precisa enviar sua chave pública para o administrador do Git, assumindo que estaremos usando um servidor SSH que requer chaves públicas. (Lembre-se que há outras formas de autenticação, como falamos antes).
- Basta copiar o conteúdo do arquivo **.pub** e enviar um email com ele.

Git

Gerando uma Chave Pública SSH

- O conteúdo da chave é semelhante a isso:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC0AywLrYotM8dPffukfkRMpG2w8Hkf3UBh6Wqdo
wPeABJ42DKvwt90990jhjDy8FjkwCloognMKzstuKOsToE713MzmMOlNSVerfgLBGs5NR
DBYunIJa67wkPRwTD6dDJEJOrexlVA28970gd4POM41imp9HJJfWxTbssakJ5S0dqKzNp
a1pK9mV01HisxBqGWZDzfUyu7knMYRrgIk1ITNiKgvA321YuUZVFz0PXCN+aQgwr/QxuO
59KvVpI57zhBIkwS4Ew1g+BmelmOL4sYVEAWoxUn9Ie5CChuH7blWlu1Nw252gOTF+VYU
BoDZwo1WUVXGgB4JABzdEB3ksu3iL JoaoJunior@notebook-jj
```

- Caso queiram ver mais detalhes sobre geração de chave SSH, visitem: https://help.github.com/articles/generating-ssh-keys