4LINUX

OPEN SOFTWARE SPECIALISTS

CURSO 525

INFRAESTRUTURA ÁGIL COM PRÁTICAS DEVOPS

Versionamento de Código com GIT

Versionamento de Código Local

Anotações

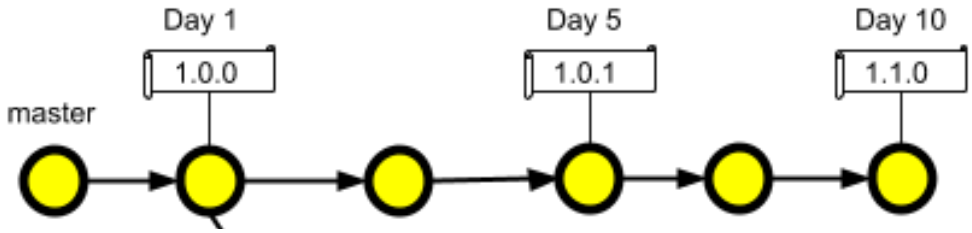
Objetivos da aula

1. O que é Versionamento de Código
2. Entendendo o GIT
3. Instalando o GIT

Anotações

O que é Versionamento de Código

Versionamento de código é o processo de se criar versões de códigos de acordo com uma linha do tempo para o andamento do desenvolvimento de melhorias, correções ou novas funcionalidades.



Se você é um designer gráfico ou um web designer e quer manter todas as versões de uma imagem ou layout (o que você certamente gostaria), usar um Sistema de Controle de Versão (Version Control System ou VCS) é uma decisão sábia. Ele permite reverter arquivos para um estado anterior, reverter um projeto inteiro para um estado anterior, comparar mudanças feitas ao decorrer do tempo, ver quem foi o último a modificar algo que pode estar causando problemas, quem introduziu um bug, quando e muito mais. Usar um VCS, normalmente, significa que se você estragou algo ou perdeu arquivos, poderá facilmente reavê-los. Além disso, você pode controlar tudo sem maiores esforços.

O que é Versionamento de Código

Para o versionamento de código, são utilizados sistemas chamados de **VCS (Version Control System)** ou **SCM (Source Code Management)**, que são softwares que têm a finalidade de gerenciar as diferentes versões no desenvolvimento de um código. Alguns exemplos de softwares de controle de versão Open-Source são:

- CVS
- Mercurial
- GIT
- SVN

Antigamente, existia muito o costume de comprar servidores para hospedagem de algum serviço, e ás vezes, o serviço não precisava de muito recurso. Porém, por conta dele estar instalado nesse servidor, você não pode mais instalar outros serviços que usem a mesma porta, você perdeu um espaço no datacenter para um serviço importante, porém que pouco usa recurso e ainda precisa pagar altíssimas contas de energia, somente para manter o serviço funcionando.

A virtualização veio para solucionar esse problema: com ela, podemos repartir os recursos de nossa máquina física em diversas máquinas virtuais, cada uma com a quantidade necessária para o serviço que possui funcionar. Por conta de sua natureza, máquinas virtuais são isoladas, só precisando se comunicar com o virtualizador, sobre o qual elas foram criadas.

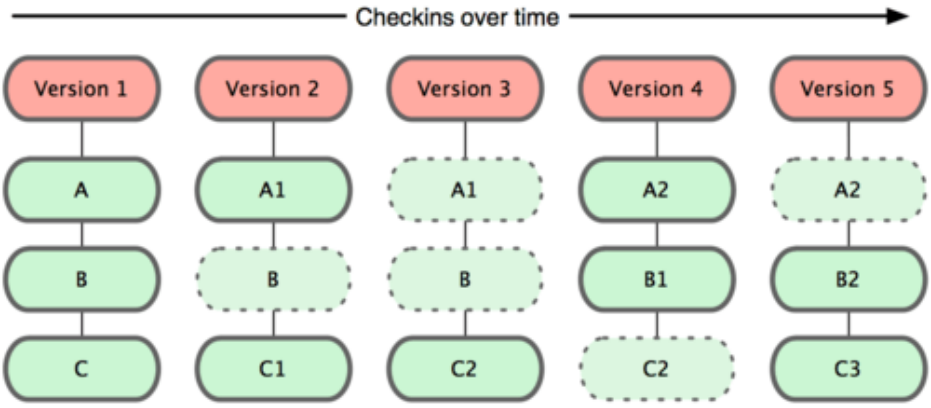
O que é o GIT

GIT é um VCS amplamente utilizado nos dias de hoje para o processo de desenvolvimento de software e versionamento de documentos.
Foi projetado e desenvolvido por Linus Torvalds para o desenvolvimento do kernel Linux e se tornou a maior ferramenta OpenSource de versionamento do mercado.

Anotações

Entendendo o GIT

O Git trabalha com dados como se fossem grupos de **snapshots** de um pequeno sistema de arquivo.

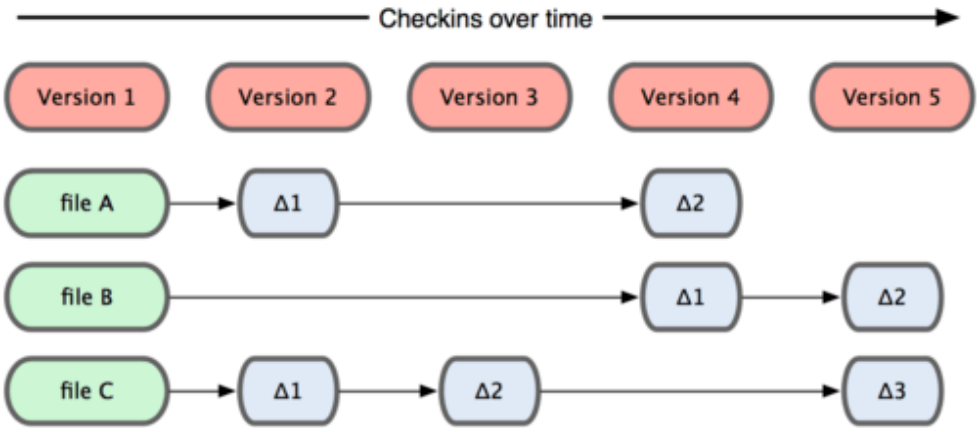


Toda vez que um usuário realiza um commit ou salva o estado atual do projeto, o Git tira uma imagem do estado de todos os arquivos e grava uma referência do estado (**snapshot**). Para ser eficiente, se o arquivo não possui nenhuma alteração, o GIT não grava os arquivos novamente, ele simplesmente cria um link para o arquivo anterior. O GIT pensa como se os arquivos e suas alterações fossem um fluxo de snapshots.

Essa arquitetura faz com que o GIT trabalhe mais como um pequeno filesystem que um sistema de versionamento comum, com ótimas ferramentas construídas por cima desse “filesystem”.

Outros VCS

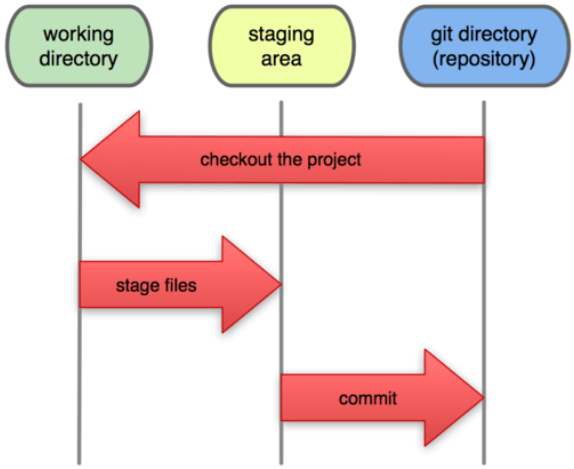
Outros sistemas costumam armazenar dados, como mudanças em uma versão inicial de cada arquivo.



A maior diferença entre Git e qualquer outro VCS (Subversion e similares inclusos) está na forma que o Git trata os dados. Conceitualmente, a maior parte dos outros sistemas armazena informação como uma lista de mudanças por arquivo. Esses sistemas (CVS, Subversion, Perforce, Bazaar, etc.) tratam a informação que mantém como um conjunto de arquivos, e as mudanças feitas a cada arquivo ao longo do tempo, conforme ilustrado na imagem acima.

GIT – Os Três Estados

Diretório de Trabalho, área de preparação e o diretório do GIT.



Essa é uma das partes mais importantes para entender o funcionamento do Git. O Git faz com que seus arquivos sempre estejam em um dos três estados fundamentais: consolidado (committed), modificado (modified) e preparado (staged).

commited: Dados são ditos consolidados, quando estão seguramente armazenados em sua base de dados local;
modified: Modificado, trata de um arquivo que sofreu mudanças, mas que ainda não foi consolidado na base de dados;
stage: Um arquivo é tido como preparado, quando você marca um arquivo modificado em sua versão corrente para que ele faça parte do snapshot do próximo commit (consolidação).

Isso nos traz para as três seções principais de um projeto do Git: o diretório do Git (git directory, repository), o diretório de trabalho (working directory) e a área de preparação (staging area).

O diretório do Git é o local onde o Git armazena os metadados e o banco de objetos de seu projeto. Esta é a parte mais importante do Git, e é a parte copiada quando você clona um repositório de outro computador.

O diretório de trabalho é um único checkout de uma versão do projeto. Estes arquivos são obtidos a partir da base de dados comprimida no diretório do Git e colocados em disco para que você utilize ou modifique.

A área de preparação é um simples arquivo, geralmente contido no seu diretório Git, que armazena informações sobre o que irá em seu próximo commit. É bastante conhecido como índice (index), mas está se tornando padrão chamá-lo de área de preparação.

O workflow básico do Git pode ser descrito assim:

1 - Você modifica arquivos no seu diretório de trabalho;
2 - Você seleciona os arquivos, adicionando snapshots deles para sua área de preparação;
3 - Você faz um commit, que leva os arquivos como eles estão na sua área de preparação e os armazena permanentemente no seu diretório Git.

Instalação do GIT

1	Instalação do GIT em distribuições baseadas em Debian/Ubuntu:
	`# sudo apt update`
	`# sudo apt install git`

2	Instalação do GIT em distribuições baseadas em Red Hat: `# sudo yum install git`
3	Verifique a instalação do GIT através do comando: `# git --version`

Anotações

Recapitulando

1. O que é Versionamento de Código
2. Entendendo o GIT
3. Instalando o GIT

Anotações

Versionamento de Código com GIT

Comandos essenciais GIT

Anotações

Objetivos da aula

1. Iniciando um repositório
2. Comandos básicos: add, commit e status
3. Clonando um repositório

Anotações

Acessando a máquina Automation

O comandos essenciais serão digitados na máquina automation. Execute os comandos abaixo para preparar a infraestrutura do curso.

1 Acesse a pasta infraagil para iniciar a aula

```
# cd infraagil
```

2 Inicie suas instâncias do Vagrant:

```
# vagrant up
```

3 Conectando ssh na máquina automation:

```
# vagrant ssh automation
```

4 Instale o Git através do seguinte comando:

```
# sudo yum install git -y
```

automation.4labs.example

Anotações

Iniciando um Repositório

Qualquer diretório pode se tornar um repositório do GIT, para isso basta executar o comando git init.

1 Crie um novo diretório:

```
# mkdir projeto
```

2 Entre no diretório:

```
# cd projeto
```

3 Inicie o repositório:

```
# git init
```

Ao digitar esse comando, será criado um diretório oculto chamado: **.git** em qualquer divisão de disco.

automation.4labs.example

Iniciando repositórios

Este comando criará um diretório oculto chamado **.git**. Esse diretório é a base de dados do Git, e constitui o repositório em si. Nele serão guardadas as informações de **commits**, **tags**, **branches**, ou seja, todas as informações referentes ao repositório.
Os comandos que aprenderemos do git manipularão o conteúdo desta pasta, por isso não precisaremos mexer diretamente nela.

Iniciando um Repositório

Algumas informações acompanharão esse processo, tais como: e-mail, nome, data, hora e mensagem de identificação. Algumas o Git gera sozinho, outras devemos informar.

1 Configure o nome do proprietário:

```
# git config --global user.name "Seu Nome"
```

2 Configure o e-mail do proprietário:

```
# git config --global user.email "seuemail@seudominio.com.br"
```

Com o comando git config, salvamos os dados em variáveis de configuração do git. Com isto, ao enviar os dados para um servidor, estas informações também serão enviadas.

automation.4labs.example

O parâmetro **--global** diz ao git que salvaremos esses valores no arquivo **~/.gitconfig**, essas configurações serão utilizadas para todos os outros projetos daquele usuário. Existe mais outro parâmetro, o **--system**, que salva as informações em **/etc/gitconfig**, sendo usado para todos os projetos de todos os usuários.

A ausência desses parâmetros gravará as informações no .**git/config**, sendo assim, válidas apenas para aquele projeto. Podemos forçar a utilização do **.git/config** passando o parâmetro --**local**.

Iniciando um Repositório

Outras configurações podem ser realizadas. Vamos configurar o editor padrão do Git que aparecerá sempre que salvarmos uma nova versão do projeto.

1. Instale o editor vim através do seguinte comando:

```
# sudo yum install vim -y
```

2. Altere o editor padrão:

```
# git config --global core.editor vim
```

3. Liste todas as configurações setadas:

```
# git config --list
```

automation.4labs.example

Anotações

Comandos Básicos

Agora que temos nossas informações já configuradas, vamos iniciar o versionamento do diretório.

1. Crie um arquivo qualquer:
```
# touch arquivo1
```

2. Adicione o arquivo para a área de stagging:
```
# git add arquivo1
```

3. Visualize o estado atual do repositório:
```
# git status
```

4. Grave a nova versão do repositório:
```
# git commit -m "Meu primeiro commit"
```

automation.4labs.example

O comando **git add** adiciona o arquivo em uma área chamada **stagging**.

Depois que temos o arquivo em stage, podemos gravar o estado dele com o comando **git commit**. Este comando salva o estado dos arquivos que estão na área de stage.

O comportamento é semelhante ao bater uma fotografia do repositório naquele exato momento. O parâmetro **-m** atribui uma mensagem ao commit, servindo para identificar mais facilmente o que foi realizado naquela alteração.

Já o comando **git status**, mostra o estado do repositório naquele momento. Trata-se de um comando bem importante que usaremos muito em nosso curso.

Comandos Básicos

Para modificar nosso último commit, vamos criar outros arquivos.

1	Crie mais 2 arquivos: `# touch arquivo2 arquivo3 arquivo4`
2	Adicione o arquivo para a área de stagging: `# git add arquivo2 arquivo3`
3	Visualize o estado atual do repositório: `# git commit -m "Commit incompleto"`
4	Adicione o arquivo4: `# git add arquivo4`
5	Envie os arquivos faltantes ao commit anterior: `# git commit --amend`

automation.4labs.example

Modificar último Commit

Ao executar o passo 3, o git retornará uma mensagem informando que os arquivos (arquivo2 e arquivo3), passaram por commit indo para o repositório local.

Para modificar o commit, deve-se adicionar o arquivo (arquivo4) no passo 4, utilizando a opção --amend para modificar o último commit, passo 5.

Clonando um Repositório

Também podemos clonar um repositório que já existe na web.

1. Clone um repositório:
```
# git clone https://github.com/leachim6/hello-world.git
```

2. Clone um repositório e dê um nome específico ao novo diretório:
```
# git clone https://github.com/leachim6/hello-world.git ola-mundo
```

3. Perceba que existem duas pastas com nomes diferentes, mas com o mesmo conteúdo:
```
# ls hello-world
# ls ola-mundo
```

automation.4labs.example

Iniciando repositórios

Este comando criará um diretório oculto chamado **.git**. Esse diretório é a base de dados do Git, constitui o repositório em si. Nele serão guardadas as informações de **commits**, **tags**, **branches**, ou seja, todas as informações referentes ao repositório.

Os comandos que aprenderemos do git manipularão o conteúdo desta pasta, por isso não precisaremos mexer diretamente nela.

Recapitulando

1. Iniciando um repositório
2. Comandos básicos: add, commit e status
3. Clonando um repositório

Anotações

Versionamento de Código com GIT

Trabalhando com branches em projetos

Anotações

Objetivos da aula

1. O que são branches
2. Criando, listando e trocando de branches
3. Merge

Anotações

O que são branches

Realizando uma associação com árvores, branches (ramos) são linhas que divergem do tronco principal de desenvolvimento. Em novos ramos, realizamos o desenvolvimento de componentes que serão combinados com o projeto principal.

Anotações

O que são branches

Criar uma branch no Git significa:

"...divergir da linha principal de desenvolvimento e continuar a trabalhar sem bagunçar essa linha principal".

Fonte: https://git-scm.com/book/pt-br/v1/Ramificação-Branching-no-Git

Anotações

O que são branches

Normalmente em ambientes de integração contínua, existem 3 branches para os projetos:

Master	Versão atual, pronta para produção.
Homolog	Versão de homologação, código sendo testado.
Develop	Branch, onde são feitos os commits.

Anotações

Criando, listando e trocando de branches

Tudo no Git é feito através de branches. A branch principal é a master, automaticamente criada sempre que um projeto git é clonado ou iniciado, a partir de um diretório existente.

1	Liste branches disponíveis: `# git branch`
2	Crie uma nova branch: `# git branch development`

3	Troque de branches:
	`# git checkout development`
	`# git checkout master`

4	Crie e troque de branch: `# git checkout -b homolog`

automation.4labs.example

No exemplo do slide, inicialmente estamos na branch master. Em seguida, utilizamos o comando **git branch** para criar uma branch chamada development. Ao executar o comando **git branch** novamente, observamos que embora o branch development tenha sido criado, ainda estamos na branch master. A branch development por enquanto, é uma cópia idêntica da branch master.

Depois, o comando git checkout nos trocou para a branch desejada. Agora, ao executar o comando git branch, verificamos a mudança para a branch development. Podemos mudar qualquer arquivo dentro da branch development, isso não afetará nada na branch master, ela sempre estará intacta.

Usando o comando git checkout -b podemos criar uma nova branch e automaticamente trocar para a branch criada.

Não há limites para essas ramificações. Podemos gerar branches a partir de outras, e com isso, gerando versões diferentes de um mesmo projeto. Utilizar branches ajuda muito quando precisamos criar uma nova feature ou corrigir algo em específico, sem modificar a branch principal.

Criando, listando e trocando de branches

Além de criar e modificar as branches, também podemos excluí-las:

1 Liste branches disponíveis:

```
# git branch
```

2 Exclua uma branch:

```
# git branch -d development
```

3 Force a remoção de uma branch:

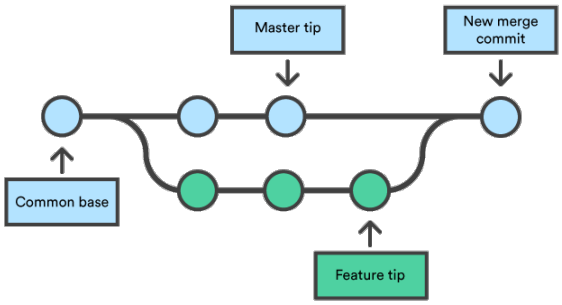
```
# git branch -D homolog
```

automation.4labs.example

Anotações

Merge

Merge é o processo realizado quando terminamos de desenvolver alguma feature e precisamos que nosso código saia de uma branch para a outra, como por exemplo de desenvolvimento para homologação.



Anotações

Merge

Vamos criar uma nova Branch para simular os Merges:

```
1  # git checkout -b develop
2  # touch arquivo06
3  # git add arquivo06
4  # git commit -m "Novo Commit"
5  # git checkout master
6  # git merge develop
```

automation.4labs.example

Sempre que criamos uma nova branch, esta inicia como uma cópia da branch atual. Tudo que for feito nessa nova branch e passar por commit, produzirá uma diferença em relação a branch copiada.

O git merge trás esses commits a frente e os aplica na branch atual.

Recapitulando

1. O que são branches
2. Criando, listando e trocando de branches
3. Merge

Anotações

Versionamento de Código com GIT

Ignorando arquivos no versionamento

Anotações

Objetivos da aula

1. Porque ignorar aquivos
2. .gitignore
3. Utilizando o .gitignore

Anotações

Por que ignorar arquivos?

Normalmente no processo de desenvolvimento de softwares, existem alguns arquivos que não gostaríamos que fossem versionados, como por exemplo arquivos compilados, arquivos com senha ou arquivos temporários que são criados por editores.

Anotações

.gitignore

Com o GIT, o controle destes arquivos fica muito mais fácil, podemos criar um arquivo chamado **.gitignore**, passar algumas opções e assim, conseguiremos dizer quais arquivos por projeto ou usuário não devem ser publicados.

Anotações

.gitignore

A maneira mais simples e fácil de gerenciar as exclusões, é criar um arquivo **.gitignore** no diretório raiz do projeto.

Os arquivos que escolheremos para ignorar, serão ignorados em todos os diretórios do projeto.

!! É importante lembrar que o arquivo é um arquivo oculto, por isto seu nome começa com o caractere ponto (.) !!

Anotações

.gitignore

O Arquivo .gitignore consegue compreender alguns tipos de declarações, tais como:

- Arquivos Específicos → `arquivo1.txt`
- Curingas → `*.txt`
- Diretórios → `/files/keys/`
- Expressões Regulares (Regex) → `arquivo{1..5}.txt`

O gitignore especifica arquivo em estado untracked que o Git deve ignorar. Qualquer arquivo que se apresente em estado tracked, continuará como tal, precisando ser removido manualmente.

Sintax arquivo gitignore
Qualquer linha vazia ou começando com “#” ou espaço, é desconsiderada pelo Git. No caso de um padrão começando com “#”, o mesmo deve ser escapado com "\", bem como espaço.

“!” — Deve ser utilizado para negar um padrão, tornando-o valido novamente. Não valendo para diretórios;

“foo/” — Caso o padrão termine em “/”, o git irá considerar somente o diretório foo;

“foo” — Nesse caso, será tratado como um padrão a ser verificado no level mais alto do diretório, onde o .gitignore se encontra;

“**/” — Casa qualquer diretório. Por exemplo: se especificado “**/foo”, será válido para “foo/” e “**/foo/bar”, sendo válido para qualquer diretório “bar” antecedido por “foo”;

“/**” — Casa todo o conteúdo dentro do diretório;

“a/**/b” — Casa qualquer conteúdo entre os diretórios existentes e entre a e b.

Utilizando o .gitignore

Agora, iremos criar nosso arquivo **.gitignore** com alguns padrões:

1. Crie o arquivo .gitignore:

```
# vim .gitignore
config.conf
```

2. Adicione o arquivo ao repositório:

```
# git add .gitignore
```

3. Realize o commit:

```
# git commit -m "Adicionado gitignore"
```

automation.4labs.example

No exemplo do slide, qualquer arquivo com o nome de **config.conf** em qualquer diretório, será ignorado no commit.

Utilizando o .gitignore

Vamos criar agora o arquivo config.conf e mais um arquivo chamado conf.txt:

1. Crie arquivos:
```
# touch config.conf conf.txt
```

2. Adicione todos os arquivos ao commit:
```
# git add --all
```

3. Verifique os arquivos:
```
# git status
```

4. Efetue o commit:
```
# git commit -m "Testando o gitignore"
```

automation.4labs.example

Anotações

Utilizando o .gitignore

Vamos editar nosso arquivo .gitignore para adicionar outros exemplos:

1

Edite o arquivo .gitignore:

```
# vim .gitignore

# Ignorar os arquivos específicos
config.conf

# Ignorar arquivos por coringa
*~
*.swap

# Ignorar diretórios
files/secret
```

automation.4labs.example

No exemplo do slide, qualquer arquivo da lista será em qualquer diretório ignorado no commit, bem como o diretório files/secret.

Utilizando o .gitignore

Após a edição do arquivo, precisamos efetuar o commit do mesmo:

1. Adicione o arquivo ao repositório:

```
# git add .gitignore
```

2. Realize o commit:

```
# git commit -m "Atualizando o gitignore"
```

automation.4labs.example

Anotações

Utilizando o .gitignore

Agora, vamos criar os arquivos e pastas que iremos excluir do commit:

1. Crie diretórios:
```
# mkdir -p files/secret
```

2. Crie arquivos:
```
# touch file~ file.swap arquivoX files/secret/secret{1..5}
```

3. Adicione os arquivos ao repositório:
```
# git add .
```

4. Verifique os arquivos:
```
# git status
```

5. Verifique os arquivos:
```
# git commit -m "Atualizando o repositorio"
```

automation.4labs.example

Ao executar o comando git status, podemos verificar que os arquivos que se encaixam no padrão descrito no gitignore não serão comitados.

Recapitulando

1. Porque ignorar aquivos
2. .gitignore
3. Utilizando o .gitignore

Anotações