Versionamento usando Git

O Git (site oficial) é um sistema de controle de versão (em inglês: version control system - VCS) distribuído, rápido e escalável, usado principalmente para o controle de versão de software. Criado em 2005 por Linus Torvalds para o desenvolvimento do kernel Linux, logo a plataforma foi adotada por diversos outros projetos. É um software livre, distribuído sob os termos da versão 2 da GNU General Public License.

Essa documentação possui 2 tópicos principais:

1. Recursos do Git, apresentação de alguns conceitos do Git e seus comandos.
2. Versionamento no Cronapp, apresentação das ferramentas *low-code* para executar os mesmos recursos Git.

Versionamento (plano FREE)

Os projetos dos usuários com o plano FREE só podem utilizar o Git interno do Cronapp, não sendo possível exportá-los para repositórios Git externos, como o Github. Além disso, contam com uma versão mais básica dos recursos de versionamento.

A documentação Versionamento (plano FREE) apresenta mais detalhes dos recursos Git para o plano FREE.

Principais recursos do Git

O Git pode ser entendido como um serviço de gestão de arquivos definido por um protocolo, possui diversos recursos e dá suporte para que múltiplas pessoas trabalhem ao mesmo tempo no desenvolvimento de um projeto.

O Cronapp possui um terminal Linux onde os comandos Git podem ser executados, porém veremos mais adiante que é possível executar os principais recursos de forma visual. Para abrir o terminal, clique no menu sanduíche, no canto esquerdo da tela, e, em seguida, no ícone destacado na figura 1.1.

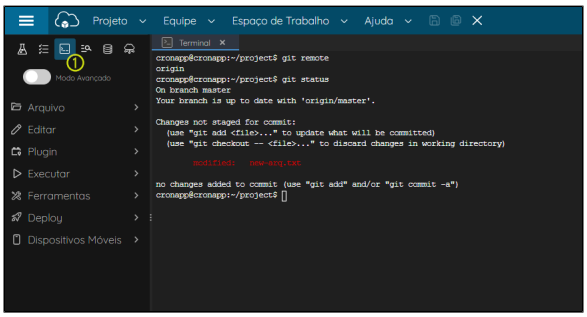

Figura 1.1 - Executando comandos Git via terminal

Repositórios

Cada desenvolvedor em um projeto no Git possui um ambiente local totalmente isolado, sendo necessário enviar e obter manualmente as atualizações. Se seu projeto estiver centralizado em um repositório remoto, cada participante deve criar um clone '*local*' desse repositório.

Nesta página

- Versionamento (plano FREE)
- Principais recursos do Git
  - Repositórios
    - Repositório local
    - Repositório Remoto
  - Uso de branchs
  - Mesclar conteúdo
  - Obter atualizações
  - Enviar atualizações
  - Ignorar conteúdo
    - Ignorar bibliotecas do Cronapp
  - Pull Request
  - Tags
  - Alterações do projeto
- Versionamento no Cronapp
  - Visão geral
  - Importar projetos
  - Exportar projetos
  - Menu de opções
    - Janela Controle de versão
    - Janela de revisões
    - Commit
    - Push with Local Tags
    - Merge
    - Tags
    - Branch
      - Novo Branch
- Autenticação com o Github
  - Autorização
  - Autorização em Organizações
  - Revogar autorização
- Autenticação com o Bitbucket
  - Autorização
  - Criar o App password

Conteúdo complementar

- Fluxo de trabalho Git flow
- Solução de conflitos do Git

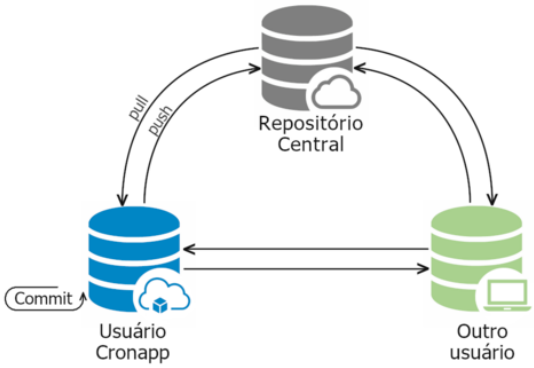

Figura 1.2 - Interação entre repositórios

Ao usar o comando `git init` dentro do diretório, o git prepara esse local para ser versionado, na prática, o git cria um subdiretório oculto com a estrutura de dados usado pelo git. Mas não se preocupe com isso, qualquer projeto criado no Cronapp já possui essa estrutura de versionamento.

```
$ git init
```

Repositório local

Diferentemente do SVN, em que o comando `git commit` envia as alterações para o repositório central, no Git, quem faz essa ação para o repositório de origem é o comando `git push`. O `git commit` tem a função de salvar localmente o estado atual (*snapshot*) dos arquivos alterados, dessa forma, é possível criar vários *snapshots* local antes de enviá-los para o repositório de origem.

O Git não cria uma cópia dos arquivos a cada nova versão, isso causaria uma enorme perda de espaço e lentidão. Ao invés disso, ele rastreia as alterações feitas e mantém o histórico de tudo.

O processo entre um *commit* e outro possui três estágios: *Working Directory*, *Staging Area* e *Repository* (Figura 1.3).

- ***Working Directory***: logo após clonar um repositório ou aplicar um `git commit`, o Git considera o *Working Directory* "limpo", ou seja, todos os arquivos estão rastreados (*Tracked*). Porém, ao adicionar um novo arquivo ou editar um existente, o Git passa a vê-lo como não rastreável (*Untracked*), isso significa que o arquivo não existia ou não está mais igual *snapshot* anterior.
- ***Staging Area***: antes de criar um novo *commit*, é necessário informar ao Git quais arquivos farão parte desse *snapshot*, assim podemos usar o comando `git status` para ver o conteúdo não rastreado ou modificado e em seguida usar o comando `git add <arquivos>` (ou `git add *` para adicionar todos) que prepara o conteúdo informado, indexando para a próxima etapa.

  ```
  git status
  git add *
  ```

- ***Repository***: neste último estágio o Git passa a rastrear todas as alterações dos arquivos informadas no estágio anterior, salvando e criando um histórico local dessas modificações. Essa ação é feita usando o comando `git commit`.

  ```
  $ git commit -m "<mensagem de resumo para o commit>"
  ```

O comando `git status` permite visualizar o estado atual de cada arquivo alterado.

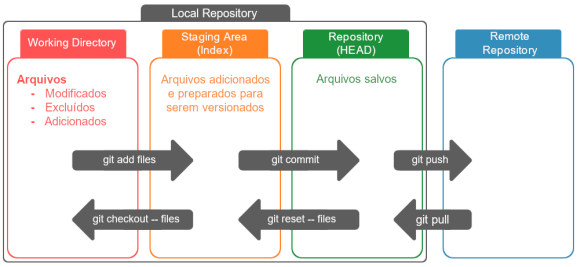

Figura 1.3 - Estágios e seus comandos

Repositório Remoto

Por ser um sistema de controle de versão distribuído, o repositório remoto é apenas um link de comunicação entre dois repositórios, onde o seu repositório é o *local* e o repositório remoto pode estar em outro domínio, normalmente nomeado como *origin*.

Podemos usar o `git clone` para obter uma cópia local de um repositório remoto.

```
$ git clone <endereço repositório remoto>
```

Como podemos ver na figura 1.2, é possível estar conectado a mais de um repositório remoto. Isso permite que você possa ter acesso ao repositório local de um colega e contribuir antes dele enviar para o repositório central.

O comando abaixo adiciona um novo repositório remoto.

```
$ git remote add <nome> <endereço repositório remoto>
```

Uso de branchs

As ramificações (*branchs*) permitem "separar" o projeto a partir de um determinado *snapshot* criando duas linhas do tempo, realizar diversas alterações e posteriormente unir (`merge`) o seu conteúdo. Assim, cada desenvolvedor pode trabalhar em um ramo próprio para, por exemplo, corrigir bugs ou criar novas funcionalidades sem impactar o fluxo principal até que o código esteja suficientemente maduro.

O primeiro comando abaixo cria uma ramificação baseada no *branch* atual, já o segundo, lista todos os *branchs* locais e remotos.

```
$ git branch <nome branch>
$ git branch -a
```

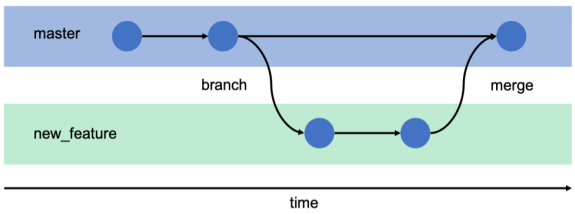

Figura 1.4 - Processo de criação, implementação e mesclagem de uma branch

No Git, o ramo principal é conhecido como *master* e a partir dele que são geradas novos *branchs*. É possível trabalhar localmente com vários *branchs*, alternando entre eles de modo que um não interfira no outro até que será feito o *merge*. Use o comando abaixo para intercalar entre os *branchs*.

```
$ git checkout <nome branch>
```

Mesclar conteúdo

O `git merge` faz o processo inverso ao `git branch`, ou seja, unifica o conteúdo do ramo informado com o *branch* atual. Durante esse processo é feito um comparativo (`git diff`) para verificar se existem diferenças entre as linhas de códigos, de modo geral essa mesclagem ocorre de forma automática, porém podem ocorrer conflito de conteúdo, onde é necessário ajustar manualmente.

```
$ git merge <ramo que será mesclado com o branch atual>
```

Os dois principais modos de mesclagem usados pelo Git são:

- **Avanço rápido**, usado quando ocorre uma situação parecida com a da Figura 1.4, em que o último *commit* do *branch* principal ocorreu por conta da bifurcação, dessa forma o Git não necessita comparar os históricos, apenas transfere o histórico da *branch* secundária para a principal.
- **Três vias**, ocorre quando há alterações nos dois fluxos, assim o Git compara o conteúdo do *co mmit* que gerou a bifurcação (base) com o conteúdo do último *commit* de cada *branch*, podendo gerar um dos quatro casos (Figura 1.5):
    - (linha 11) O conteúdo base é igual ao conteúdo dos dois *branchs*, dessa forma não houve alterações e o conteúdo continua igual;
    - (linha 20) O conteúdo base é igual a apenas um dos *branchs*, houve alteração e o conteúdo diferente (mais atual) será adicionado;
    - (linha 27) O conteúdo só existe em um dos *branchs* bifurcados, foi inserido um novo conteúdo que será adicionado;
    - (linha 71) O conteúdo dos dois *branchs* são diferentes da base, o Git informa o conflito que deve ser resolvido manualmente.

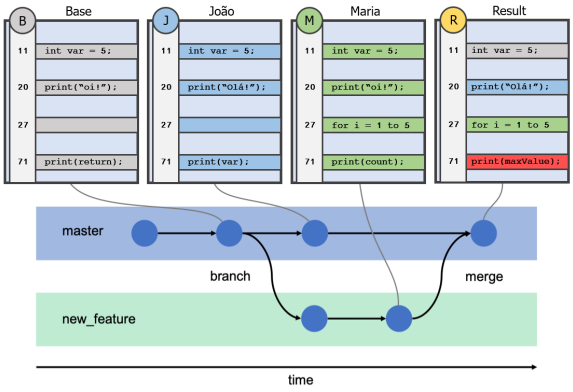

Figura 1.5 - Situações que podem ocorrer durante o merge

Obter atualizações

O git possui dois comandos para obter novos conteúdos de um repositório:

- O `git pull` baixa as atualizações e em seguida mescla o conteúdo com o do repositório local automaticamente.

    ```
    $ git pull <nome do repositório remoto> <nome branch>
    ```

- O `git fetch` baixa o conteúdo sem mesclar, assim é possível analisar primeiro se as atualizações estão seguras e só em seguida aplicar o *merge* manualmente.
  Abaixo os comandos para baixar e verificar as diferenças entre o conteúdo do *branch* atual e o baixado:

    ```
    $ git fetch
    $ git diff master Origin/master
    ```

Enviar atualizações

Após finalizar e "commitar" uma atividade em seu repositório local, podemos usar o comando `git push` para enviar o seu conteúdo para o repositório remoto. Porém, é recomendado que primeiro seja baixado o conteúdo atual do repositório remoto, resolver os possíveis conflito com a sua versão local e só então enviar.

```
$ git push
```

Em seguida o Git solicitará o seu usuário e senha do repositório remoto.

Ignorar conteúdo

Não é interessante que arquivos compilados, de cache e outros sejam enviados para os repositórios remotos, assim, podemos utilizar o arquivo `.gitignore` para informar ao Git o que não deve ser rastreado. Esse arquivo costuma ficar na raiz do projeto, contemplando toda a estrutura do projeto.

Cada linha do `.gitignore` especifica um padrão, então podemos configurar padrões como: todos os arquivos de um subdiretório, todos os arquivos com uma determinada extensão, todos os arquivos que iniciam com uma sequência determinada de caracteres e outros.

Para mais detalhes sobre os padrões, acesse a documentação do .gitignore.

Ignorar bibliotecas do Cronapp

Os projetos *web* e *mobile* possuem o diretório `node_modules`, essa pasta é utilizada pelo Cronapp para armazenar diversas bibliotecas necessárias apenas durante o período de desenvolvimento do seu projeto. Por isso, não recomendamos realizar alterações manuais nesse diretório, pois essas alterações serão sobrescritas nas próximas atualização do Cronapp. Também não recomendamos versionar esse diretório em seu repositório remoto git, afim de melhorar o desempenho ao usar e recompilar o seu projeto e evitar manter conteúdos desnecessários em seu repositório remoto.

Em projetos criados após a versão 2.8 do Cronapp, esse diretório já vem listado no arquivo `.gitignore` e qualquer atualização dentro do diretório `node_modules` é automaticamente desconsiderada ao realizar as ações de commit e push do git.

Se o seu projeto for versionado, verifique se a pasta `node_modules` já está sendo ignorada. Para isso, habilite a opção Modo Avançado, abra o arquivo `.gitignore` na raiz do projeto (destaque 1 da figura 1.6) e verifique se possui o termo "node_modules" (2). Caso não possua o termo "node_modules", adicione-o na última linha (2) e salve em seguida (para mais detalhes sobre o **gitignore**, acesse a sua do cumentação). Após isso, faça um commit e push do seu projeto.

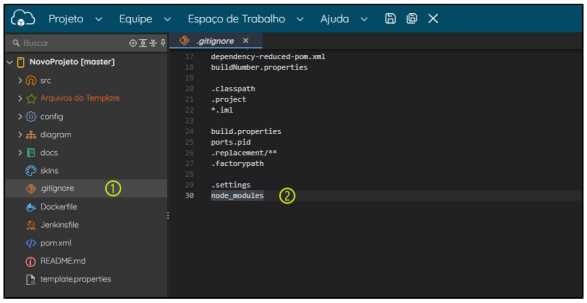

Figura 1.6 - Ignorando os diretórios node_modules nos projetos mobile e web

Após os passos acima, os diretórios `node_modules` do projeto *web* (Endereço: `src/main/webapp/`) e do projeto *mobile* (Endereço: `src/main/mobileapp/www/`) serão exibidos em cinza (destaque 1 da Figura 1.7), informando que não serão mais rastreados pelo git.

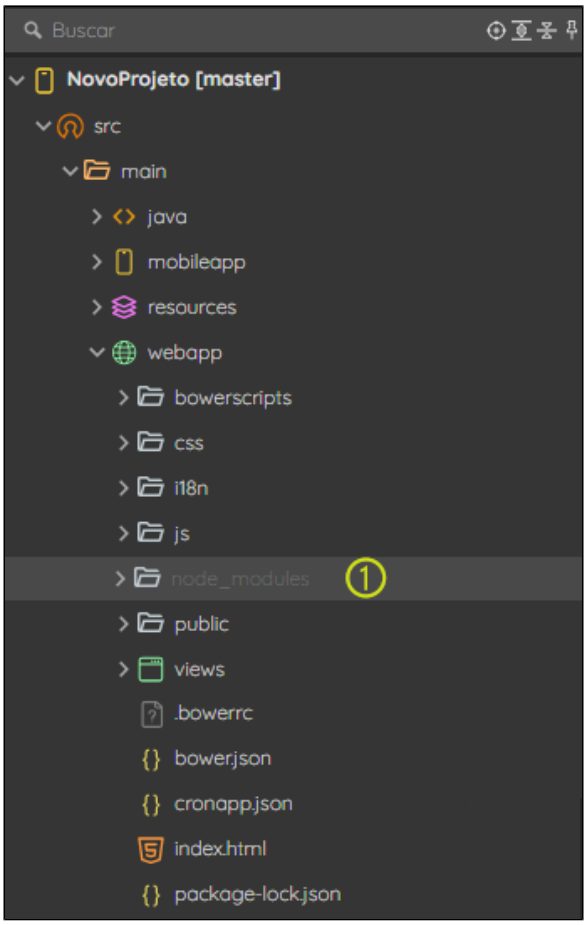

Figura 1.7 - Diretório não rastreado pelo git

Caso já tenha enviado esses diretórios para o seu repositório remoto git, recomendamos excluir esses diretórios diretamente por lá.

Pull Request

Uma solicitação *Pull* é um mecanismo muito utilizadas em ambientes corporativos e normalmente já vem integrado em sistemas com interface, como o Github, GitLab, etc. Ao inicializar uma *pull request*, é possível indicar equipes ou pessoas para validar o conteúdo a ser mesclado, nesse processo podemos discutir e revisar as possíveis alterações com os colaboradores, adicionando novos *commits* antes que as alterações sofram *merge* no *branch* base.

Requisições *Pull* normalmente trabalham em conjunto com o *GitFlow* e geralmente ocorre o seguinte processo:

1. O desenvolvedor cria um ramo em seu repositório local e atua em uma nova funcionalidade ou correção de bug.
2. Após concluir a atividade, o desenvolvedor envia o *branch* para um repositório que todos da equipe tenham acesso.
3. Em seguida é criado o *Pull Resquest*.
4. A equipe revisa o conteúdo, discute e realiza alterações, se for necessário.
5. Por fim, o mantenedor do projeto mescla o conteúdo no repositório central e fecha a Requisição *Pull*.

Tags

As tags no Git são utilizadas para referenciar um momento específico na linha do tempo do seu projeto, normalmente estão muito atreladas ao lançamento de versões (v1, v1.1, etc). É uma boa prática utilizar as tags no *branch* principal para marcar os pontos de versão, isso será muito útil para facilitar o rastreio do que foi lançado, aplicar correções, voltar versões etc. Hoje muitos sistemas utilizam as tags para controlar versões, como por exemplo o node. Acesse a documentação oficial para mais detalhes sobre as Tags.

Utilize o comando abaixo para criar um marcador e informe em "<nome da tag>" um identificador semântico, ex,: "v1.0.4" ou "v1.0.4-rc".

```
git tag -a <nome da tag> -m "<comentário sobre a versão>"
```

O envio do marcador para o repositório remoto é feito a partir do comando:

```
git push origin <nome da tag>
```

Para listar todos os marcadores já criados, use o comando abaixo:

```
git tag
```

Alterações do projeto

É possível consultar a qualquer momento, inclusive em tempo de desenvolvimento, as mudanças geradas no projeto por você ou outros desenvolvedores. O comando `git log` possibilita visualizar as mudanças ocorrida em qualquer parte do projeto, contendo observações, datas, versões, histórias de usuários e autores responsáveis pelas alterações.

Abaixo segue alguns exemplos de uso. Acesse a documentação oficial para mais detalhes.

Para listar todos os *commits* com sua descrição, autor e data.

```
git log
```

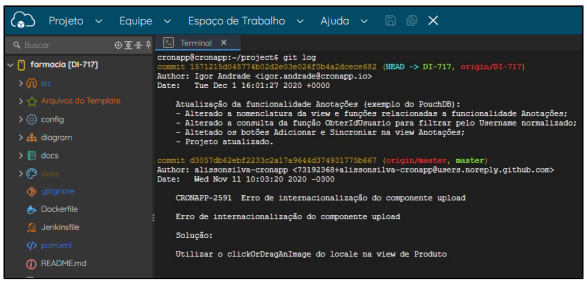

Figura 1.8 - Identificando as mudanças do projeto

Para listar as mudanças ocorridas em um arquivo específico.

```
git --no-pager log --stat -n<número de alterações> -- <endereço do arquivo>
```

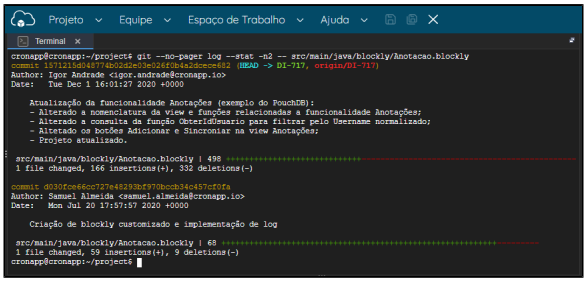

Figura 1.9 - Identificando as últimas mudanças em um arquivo específico

---

Versionamento no Cronapp

O Cronapp possui um ambiente preparado para facilitar o uso do Git em seus projetos. Mostraremos como executar os principais comandos Git de forma visual.

Visão geral

Ao abrir um projeto no Cronapp é possível visualizar algumas informações sobre o seu versionamento: o nome do *branch* ativo é exibido ao lado do nome do projeto na raiz de arquivos (destaque 2 da figura 2.1), a quantidade de versões locais não enviadas para o repositório remoto é exibido ao lado do *branch* (destaque 1 da figura 2.1) e os arquivos não rastreados pelo Git ficam com cores diferenciadas para informar sua situação atual:

- **Verde**: adicionado ou renomeado;
- **Laranja**: criado localmente;
- **Azul**: atualizado;
- **Cinza**: removido;
- **Roxo**: ausentes;
- **Vermelho**: arquivo com conflito.

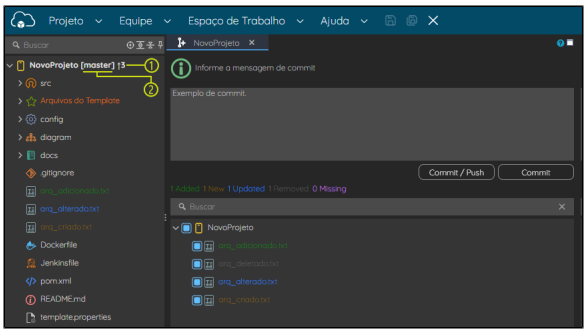

Figura 2.1 - Árvore de arquivos e janela para realizar o commit e push ou só commit

Importar projetos

Para clonar um projeto de um repositório Git para o Cronapp, crie um novo projeto e, na janela **Novo Projeto** (Figura 2.2), siga os passos abaixo.

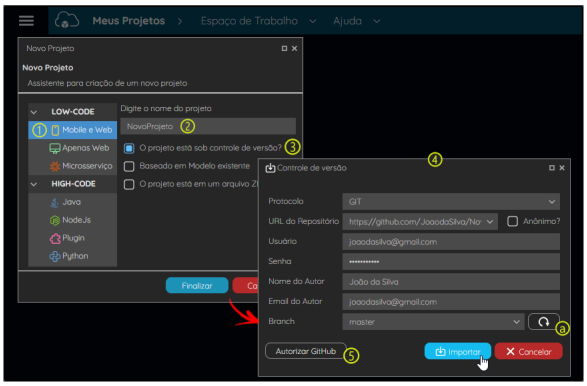

Figura 2.2 - Importar projeto de um repositório Git

1. **Tipo do projeto**: selecione o tipo do projeto a ser clonado.
    - **Low-code**: projeto Cronapp Web, mobile e servidor ou web e servidor e microsserviço.
    - **High-code**: projeto nas plataformas Java, NodeJs, Python ou Plugin Cronapp.
2. **Nome do projeto**: informe um nome para o projeto local, não é necessário ser o mesmo nome da versão remota.
3. **Origem**: selecione a opção "O projeto está sob controle de versão?" e clique em **Finalizar** para abrir a janela Controle de versão,
4. **Janela Controle de versão**: informe os dados do repositório Git a ser clonado.
    - **Protocolo**: Atualmente o Cronapp só tem suporte nativo ao protocolo Git.
    - **URL do Repositório**: endereço HTTP do repositório a ser clonado. Endereços de projetos Git sempre terminam com a extensão .git.
    - **Anônimo?**: marque essa opção para baixar repositórios públicos sem informar seu usuário e senha. Esses dados poderão ser requisitados ao fazer o `push`.
    - **Usuário**: seu usuário do serviço Git.
    - **Senha**: sua senha do serviço Git.
    - **Nome do Autor**: informado no conteúdo atualizado por você.
    - **Email do Autor**: informado no conteúdo atualizado por você.
    - **Branch**: selecione o ramo que será importado, posteriormente é possível baixar as demais *branchs* do projeto.
        a. **Recarregar**: atualiza a caixa de seleção do *branch*.
5. **Autorizar GitHub**: ao preencher o campo "URL do Repositório" com um domínio do GitHub, esse botão será exibido para logar com sua conta do GitHub. Acesse o tópico Autenticação com o Github para mais detalhes.

Exportar projetos

Para exportar um projeto no Cronapp para o Git, é preciso primeiro criar um repositório em um serviço Git, como Github, GitLab, Bitbucket ou outro qualquer. Com o endereço do repositório, basta seguir os passos abaixo.

Ao criar um repositório em serviços de terceiros, normalmente é dada a opção de já incluir o arquivo README.md, que é usado para apresentar informações sobre o projeto. Recomendamos não incluir esse arquivo para evitar conflitos, pois os projetos criados no Cronapp já possuem esse arquivo.

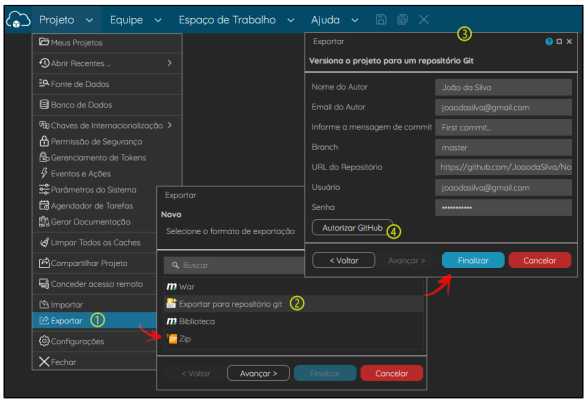

Figura 2.3 - Exportar projeto para um repositório Git

1. **Menu Projeto**: clique no menu "Projeto" e selecione a opção "Exportar" para abrir a janela que configura o modo de exportação.
2. **Janela de seleção de formato**: selecione a opção "Exportar para repositório git" e clique em **Avançar**.
3. **Janela de configurações do repositório**: preencha os dados solicitados para enviar seu projeto para o repositório informado.
    - **Nome do Autor**: informado no conteúdo atualizado por você.
    - **Email do Autor**: informado no conteúdo atualizado por você.
    - **Informe a mensagem de commit**: esse processo fará o primeiro commit, assim, informe uma mensagem.
    - **Branch**: informa a *branch* que será utilizada.
    - **URL do Repositório**: endereço HTTP do repositório. Endereços de projetos Git sempre terminam com a extensão .git.
      Caso seja um repositório Github, um botão aparecerá para realizar a autorização via GitHub.
    - **Usuário**: seu usuário do serviço Git.
    - **Senha**: sua senha do serviço Git.

4. **Autorizar GitHub**: ao preencher o campo "URL do Repositório" com um domínio do GitHub, esse botão será exibido para logar com sua conta do GitHub. Acesse o tópico Autenticação com o Github para mais detalhes.

Ao final, clique em **Finalizar**. É possível que apareça uma janela de confirmação (Figura 2.4) caso tenha conteúdo, a *branch* master será sobrescrita. Clique em **Sim** para continuar e aguarde a mensagem "Operação efetuada com sucesso.".

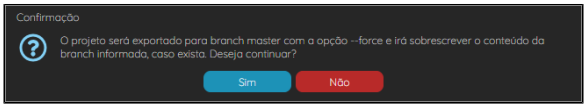

Figura 2.4 - Janela de confirmação

Menu de opções

Após versionar um projeto, todas as ações *low-code* do Git devem ser feitas através do menu **Equipe** (fig ura 2.5).

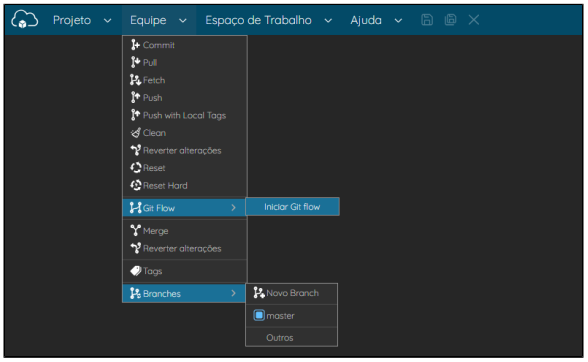

Figura 2.5 - Lista de ações do menu Equipe

Recursos disponíveis no menu **Equipe**:

- **Commit**: abre a janela que permite selecionar os arquivos que farão parte do *commit*, também é possível enviar para o repositório remoto através da opção *commit/push*.
- **Pull**: baixa e mescla as atualizações do repositório remoto.
- **Fetch**: baixa, porém não mescla as atualizações do repositório remoto. Uma janela com os arquivos a ser atualizados durante o *merge* será exibida.
- **Push**: envia para o repositório remoto todas as versões ainda não enviadas.
- **Push with Local Tags**: envia para o repositório remoto as *tags* locais que ainda não foram enviadas.
- **Clean**: essa operação removerá todos os arquivos criados ou adicionados e que ainda não foram rastreados pelo Git, ou seja, ainda estão no estágio *Working Directory* (Figura 1.3).
- **Reset**: mantêm as alterações que estão atualmente nos estágios *Working Directory* e *Staging* A rea e retorna o ponteiro HEAD para antes do último *commit*.
- **Reset Hard**: exclui as alterações que estão nos estágios *Working Directory* e *Staging* Area e retorna o ponteiro HEAD para logo após o penúltimo *commit*.
- **Git Flow**: ações usadas para o Git flow.
    - **Iniciar Git flow**: cria um *branch* develop e prepara o projeto para trabalhar com Git flow.
- **Merge**: abre uma janela para mesclar o conteúdo do *branch* selecionado no *branch* atual.
- **Reverter alterações**: abre uma janela onde é possível selecionar quais arquivos retornarão ao status do último *commit*.
- **Tags**: permite criar e recarregar pontos de *tags*.
- **Trocar Branch**: realiza ações relacionadas aos *branchs*.
    - **Novo branch**: abre a janela para criar novo *branch*.
    - **Lista de branch**: exibe a lista dos *branchs* do projeto, selecione outro *branch* para carregá-lo.
    - **Outros**: abre uma janela onde é possível realizar diversas ações com os *branchs* locais e remotos.

Você encontrará ações reduzidas ao abrir o menu de contexto de um arquivo ou diretório.

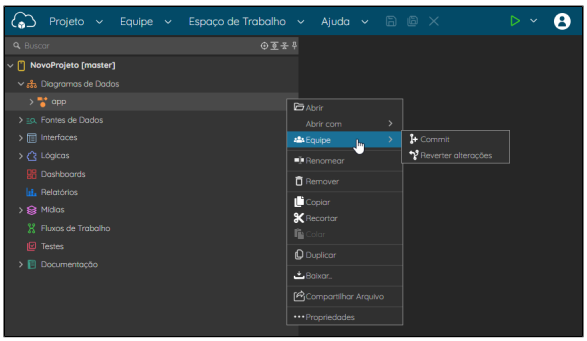

Figura 2.5.1 - Menu de contexto de arquivos ou diretórios

- **Commit**: abre a janela que permite selecionar os arquivos para aplicar um *commit* ou um *commit* e *push* em seguida.
- **Reverter alterações**: abre uma janela onde é possível selecionar quais arquivos retornarão ao status do último *commit*.

Janela Controle de versão

As informações do repositório e usuário do *Git* podem ser visualizadas por meio da janela de **Controle de versão** (destaque 2 da figura 2.6). Essa janela está disponível a partir do menu do sistema **Equipe** ou na janela de Configurações do Projeto (destaque 1). Com exceção do campo **URL do Repositório**, todos os outros campos podem ser atualizados.

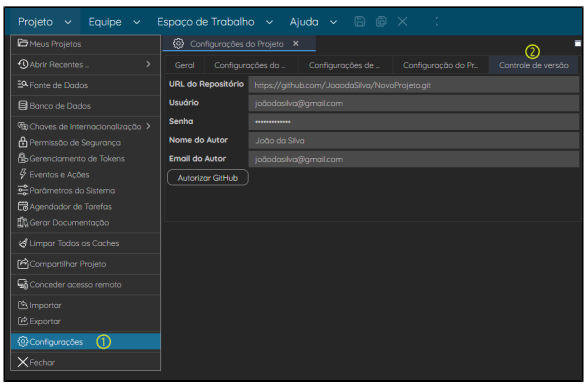

Figura 2.6 - Acessando o Controle de versão

Janela de revisões

Caso as ações *merge ou pull* apresentem conflito em pelo menos um dos arquivos, uma janela aparecerá informando o problema e os nomes desses arquivos (figura 2.7), será necessário que você resolva esse conflito antes de tentar executar a ação novamente.

Após fechar a mensagem de conflito, os arquivos com problema aparecerão destacados em vermelho, dê um duplo clique para abrir a janela de revisão de conflitos.

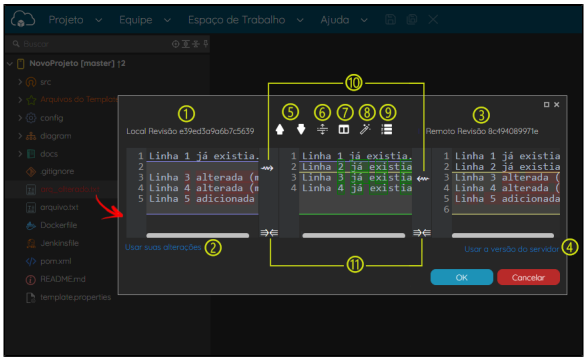

Figura 2.7 - Detalhes da janela de revisões dos conflitos

1. **Revisão local**: conteúdo do arquivo local e o ID do último *commit*.
2. **Usar suas alterações**: todos os trechos de código ainda em conflito serão resolvidos com base no código local.
3. **Revisão remota**: conteúdo do arquivo remoto e o ID do último *commit*.
4. **Usar a versão do servidor**: todos os trechos de código ainda em conflito serão resolvidos com base no código remoto.
5. **Navegação**: navega entre os blocos de códigos diferente.
6. **Expandir / recolher trechos inalterados**.
7. **Agrupar / desagrupar seções alteradas**.
8. **Destacar / ocultar diferenças**.
9. **Exibir / ocultar números de linhas**.
10. **Aplicar trecho**: clique na seta da esquerda para aplicar o trecho de código da revisão local, já a seta da direita aplicará o trecho remoto.
11. **Bloquear rolagem em conjunto**: se ativo, ao rolar o código de uma das colunas, as outras duas rolam juntas.

Commit

A janela de *commit* permite adicionar uma mensagem, selecionar os arquivos que serão adicionados na próxima versão e dá a opção para apenas *commit* ou "commitar" e enviar o conteúdo para o servidor remoto.

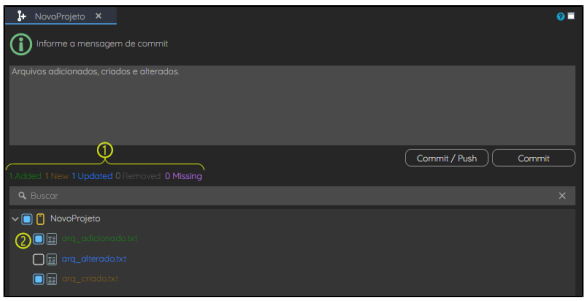

Figura 2.8 - Janela usada para criar versões

1. **Quantidade de arquivos** adicionados, novos, atualizados, removidos e desaparecidos.
2. **Checkbox**: selecione apenas os arquivos que farão parte da nova versão.

Ao dar um duplo clique sobre um dos arquivos que já foram rastreados anteriormente pelo Git, é possível abrir a janela de revisão com o conteúdo antigo e o novo. Acesse o tópico sobre a janela de revisão para mais detalhes.

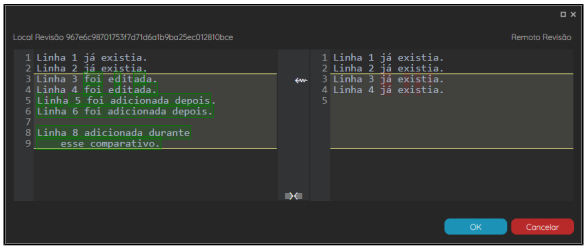

Figura 2.9 - Janela de revisão usada pra analisar as alterações do arquivo

Push with Local Tags

A ação **Push with Local Tags** será realizada apenas se a *tag* for criada no repositório local e não tiver sido enviada para o repositório remoto. Essa funcionalidade é útil, por exemplo, ao criar uma *tag* utilizand o o terminal. No entanto, ao utilizar a ferramenta de *tags* do Cronapp, o *push* será feito automaticamente.

As *tags* exibidas na imagem abaixo (Figura 2.10) foram criadas localmente via terminal. Para fazer o *pus h* de uma ou mais *tags*, você deve abrir a janela de **Push with Local Tags** e selecionar a *tag* que deseja enviar para o repositório remoto, em seguida clique no botão **Push**.

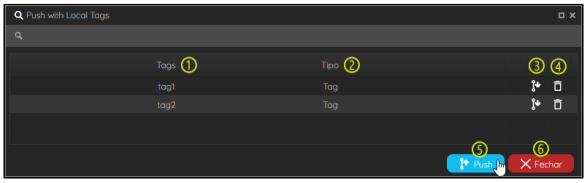

Figura 2.10 - Janela de push de tags locais

1. **Tags**: lista os nomes das tags.
2. **Tipo**: informa o tipo do elemento da lista.
3. **Checkout**: cria um novo branch a partir da tag selecionada.
4. **Remover**: exclui a tag selecionada.
5. **Push**: envia a tag para o repositório remoto.
6. **Fechar**: fecha a janela de tags.

Merge

Para aplicar o merge, você deve estar no *branch* que deseja atualizar e selecionar o *branch* que deseja obter o novo conteúdo. Se não houver conflitos, será exibido uma janela com os arquivos atualizados, caso contrário, aparecerá uma janela de erro com os arquivos em conflito. Após clicar em OK, os arquivos em conflito estarão em vermelho, dê um duplo clique para abrir a janela de revisão e resolver o conflito.

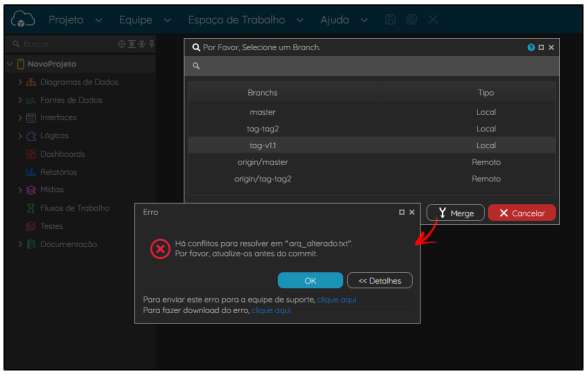

Figura 2.11 - Mensagem de conflito durante o merge

Tags

A janela Tags exibe a lista de tags do projeto e permite criar ou gerar um branch a partir de uma tag.

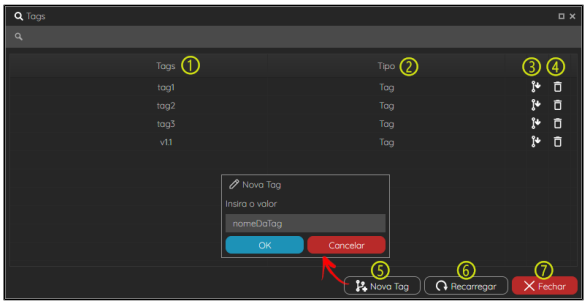

Figura 2.12 - Janela com a lista dos branchs

1. **Tags**: lista os nomes das tags.
2. **Tipo**: informa o tipo do elemento da lista.
3. **Checkout**: cria um novo branch a partir da tag selecionada.
4. **Remover**: exclui a tag selecionada.
5. **Nova Tag**: abre a janela de criação de tags.
6. **Recarregar**: verifica se existem novos *branchs* remotos e recarrega a lista.
7. **Fechar**: fecha a janela de tags.

Branch

A opção **Outros** no submenu **Trocar Branch** abre uma janela onde é possível realizar algumas ações em *branchs* locais, remotas e até *tags*.

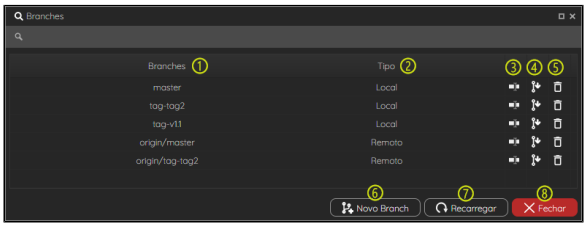

Figura 2.13 - Janela com a lista dos branchs

1. **Branches**: lista os nomes das *branches*.
2. **Tipo**: informa o tipo do elemento da lista.
3. **Renomear Branch**: renomeia a *branch* selecionada.
4. **Checkout**: recarrega o *branch* selecionado, essa ação pode ser feita com um duplo clique sobre o nome do *branch*.
5. **Deletar Branch**: exclui a branch selecionada.
6. **Novo Branch**: abre a janela para criar novo *branch*.
7. **Recarregar**: verifica se existem novos *branchs* remotos e recarrega a lista.
8. **Fechar**: fecha a janela.

Novo Branch

Ao selecionar essa opção, uma janela será aberta permitindo definir um nome para o novo *branch* e algumas ações.

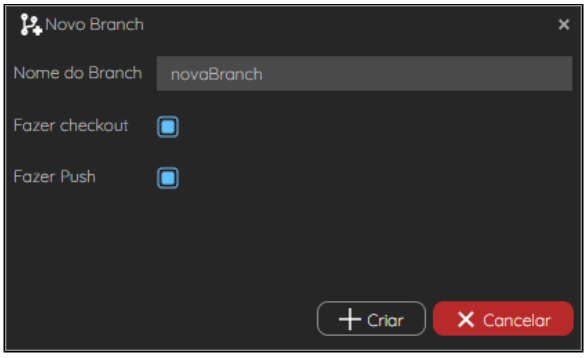

Figura 2.14 - Criando novo ramo

- **Fazer checkout**: o novo *branch* é carregado e passa a ser o *branch* atual.
- **Fazer push**: o novo ramo é enviado automaticamente para o repositório remoto.

Autenticação com o Github

O Github possui uma política de autenticação diferente da maioria dos repositórios Git. Ao invés de utilizar seu usuário e senha do Github para permitir que o Cronapp, ou outra aplicação, acesse seu repositório remoto, agora é necessário autorizar quais aplicativos poderão ter acesso. Após permitir, um token OAuth exclusivo é gerado e o aplicativo terá acesso ao repositório remoto.

Esse procedimento é necessário apenas na primeira vez ou após revogar a autorização do Cronapp. Após isso, todos os projetos que você criar no Cronapp terá a mesma permissão.

Todos os usuários do Cronapp com projetos já criados e versionados no Github precisarão fazer a mudança das credenciais. Para fazer isso, acesse Projeto > Configurações e vá na aba Controle de Versão e siga os passos abaixo.

Autorização

Os passos a seguir podem ser executados em todas as janelas de autenticação do Git dentro do Cronapp:

- Ao exportar (Figura 3.1) ou importar um projeto.
- Na aba **Controle de versão** nas Configurações do Projeto.
- Na janela **Controle de versão**, acessível através do menu de contexto do projeto.

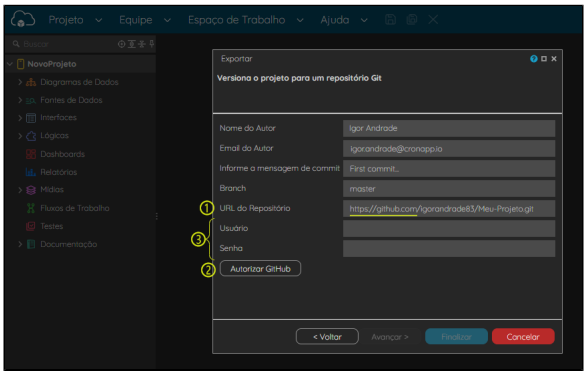

Figura 3.1 - Janela de autenticação do Git ao exportar projeto

Ao informar um endereço com o domínio do "`github.com`" no campo **URL do Repositório** (destaque 1), o Cronapp exibirá o botão **Autorizar GitHub** (destaque 2). Clicando nele, uma nova aba em seu browser será aberta para logar com sua conta no GitHub (Figura 3.2).

Não é necessário informar os campos **Usuário** e **Senha** (destaque 3), esses campos serão preenchidos automaticamente após autorizar o Cronapp no Github. Porém, caso possua um token pessoal, é possível informar seu **usuário** e **token pessoal** e, em seguida, **Finalizar**, sem precisar gerar um token em **Autorizar GitHub**.

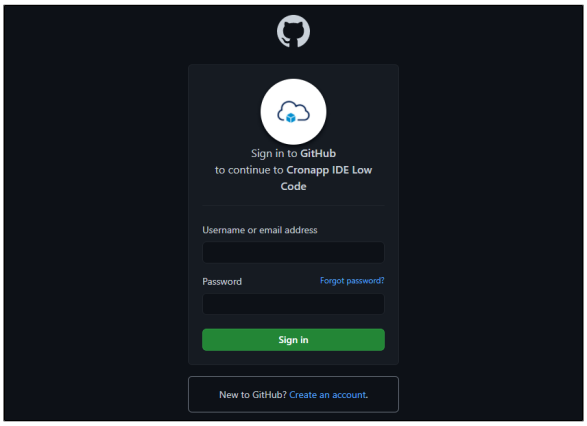

Figura 3.2 - Logue com sua conta no GitHub

Após logar, você será direcionado para a página onde poderá autorizar o Cronapp. Clique no botão **Auth orize CronApp** (destaque 1 da figura 3.3). Observe que, se você for membro de mais de uma organização, você terá que clicar no botão **GRANT** de cada uma para permitir manipular/criar repositórios nelas.

Acesse a documentação do Github para mais detalhes sobre autorização de aplicativos OAuth.

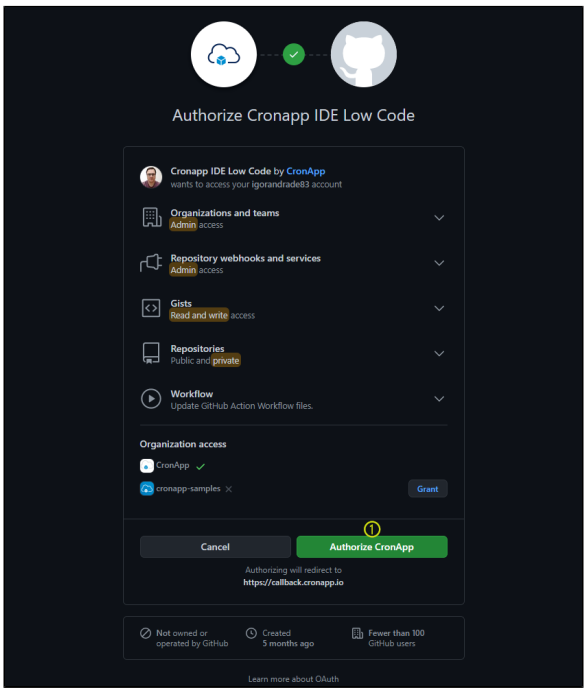

Figura 3.3 - Autorize o Cronapp

Aplicada a autorização, a aba do navegador em que estava aberto o GitHub será fechada e você retornará para a aba do Cronapp. O campo **Usuário** será automaticamente preenchido com o seu usuário do GitHub e a **Senha**, com o token gerado (destaque 1 da figura 3.4). Por fim, clique no botão **Fi nalizar**.

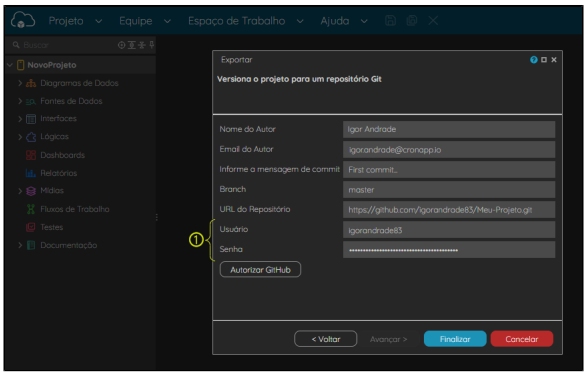

Figura 3.4 - Os campos Usuário e Senha são preenchidos automaticamente após autorizar o Cronapp em seu GitHub

Autorização em Organizações

Se o repositório pertence a uma organização GitHub, será necessário que o administrador dessa organização autorize o Cronapp. Para isso, acesse as configurações da organização no GitHub, clique em **Trird-party access** (destaque 1 da figura 3.5) e em seguida, clique no ícone de edição da aplicação **Cronapp IDE Low Code** (2). Na tela seguinte, clique no botão "Grant access" para mudar o status de "Denied" para "Approved".

Uma vez autorizado, os usuários poderão acessar os dados da organização através do Cronapp.

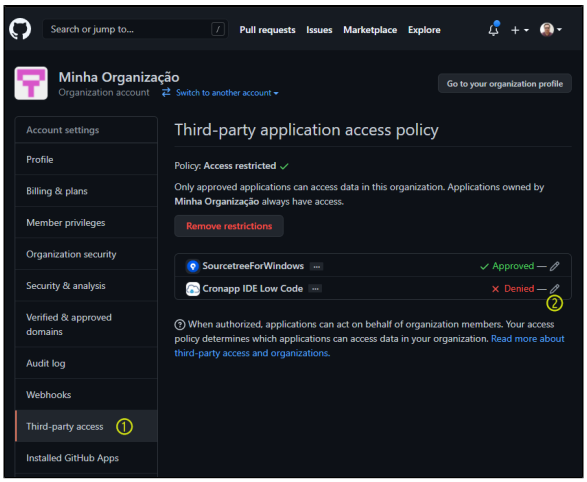

Figura 3.5 - Autorizar o Cronapp dentro da Organização GitHub

Acesse a documentação do GitHub para mais detalhes sobre restrições de acesso aos dados da organização.

Revogar autorização

Os passos a seguir mostram como revogar a autorização dada ao Cronapp em seu GitHub.

Logue em sua conta do GitHub e no menu do usuário (destaque 1 da figura 3.6) clique em **Settings** (2).

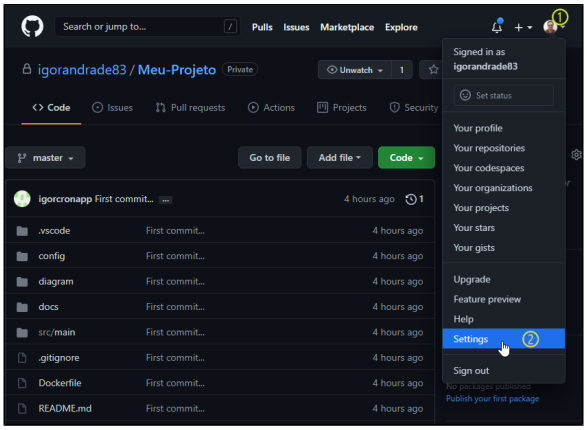

Figura 3.6 - Acesso as configurações da sua conta no GitHub

Na próxima tela, clique em **Application** (destaque 1 da figura 3.7) no menu lateral e depois na aba **Auth orized OAuth Apps** (2). Na lista de aplicativos, clique no ícone "**...**" (3) do aplicativo "Cronapp IDE Low Code" e depois em **Revoke** para abrir um pop-up confirmando a ação, por fim, clique no botão **I understand, revoke access** para revogar a autorização.

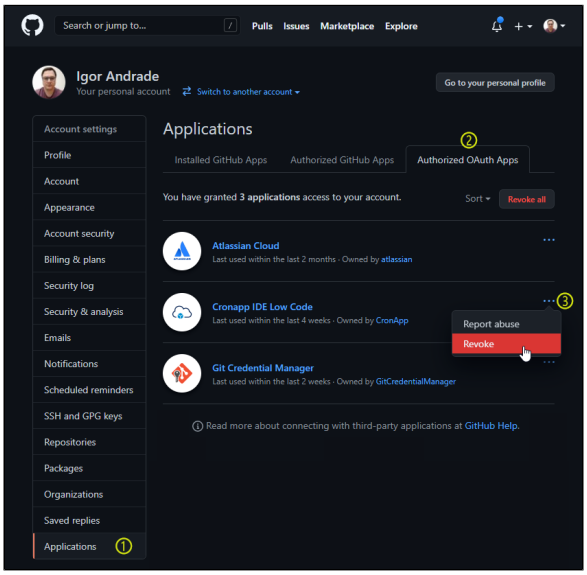

Figura 3.7 - Revogando a autorização do Cronapp ao seu GitHub

Autenticação com o Bitbucket

O Bitbucket mudou a sua política de autenticação via API. Agora, ao invés de utilizar seu usuário e senha do Bitbucket para permitir que o Cronapp, ou outra aplicação, acesse seu repositório remoto, é necessário criar uma senha do aplicativo e definir as restrições dessa senha.

Esse procedimento é necessário apenas na primeira vez ou após revogar a senha criada. Após criar a senha, todos os projetos que você utilizar essa senha terão as mesmas restrições em relação ao repositório (veja mais detalhes na documentação oficial).

Todos os usuários do Cronapp com projetos já criados e versionados no Bitbucket precisarão fazer a mudança das credenciais. Essa atualização pode ser feita na janela de **Controle de versão** (Figura 4).

Autorização

Os passos a seguir são necessários para todas as janelas de autenticação do Git dentro do Cronapp:

- Ao exportar ou importar um projeto;
- Na aba **Controle de versão** nas Configurações do Projeto;
- Na janela **Controle de versão**, acessível através do menu de contexto do projeto (Figura 4).

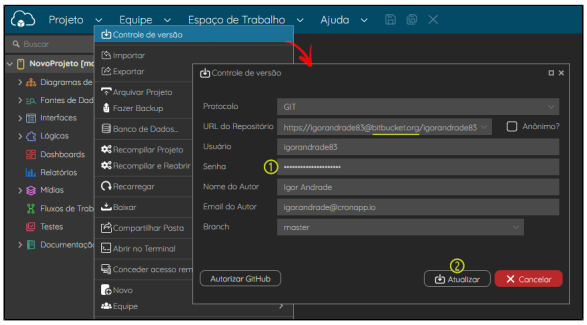

Figura 4 - Janela de autenticação do Git ao exportar projeto

1. Informe a senha gerada (destaque 1 da figura 4.4) nas configurações do seu Bitbucket (**App passwords**).
2. Clique no botão **Atualizar**.

Criar o App password

Para realizar esse procedimento, é necessário logar com seu usuário e senha da conta no Bitbucket. Após logar, acesse o menu da conta (destaque 1 da figura 4.1) e em seguida clique em **Personal settings** (destaque 2).

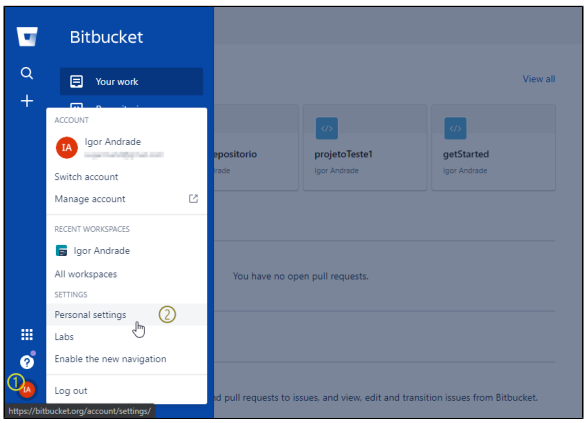

Figura 4.1 - Acesso as configurações do usuário

Na área das configurações pessoais, clique em **App passwords** (destaque 1 da figura 4.2) e depois no botão **Create app password** (destaque 2 da figura 4.2).

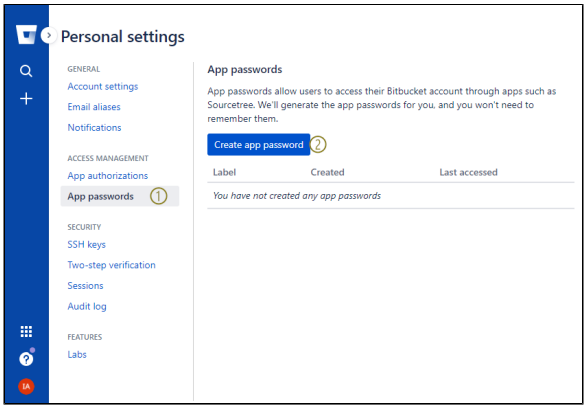

Figura 4.2 - Acesso a área de criação de novas senhas

Informe um nome para essa senha (destaque 1 da figura 4.3) e as permissões de acesso (2). Ao final, clique no botão **Create** (3).

Dependendo do nível de restrição que configurar nesta área, alguns recursos podem não ter permissão no Cronapp.

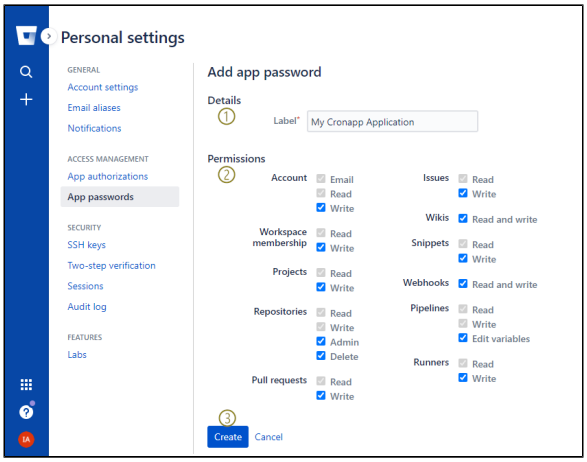

Figura 4.3 - Configurando as permissões da senha

Após confirmar a criação (destaque 3 da figura 4.3), o Bitbucket irá gerar uma senha com as restrições definidas na sua criação (destaque 1 da figura 4.4).

Anote essa senha com cuidado, pois não será possível visualizá-la novamente. Caso esqueça, será necessário gerar uma nova (veja mais detalhes na documentação oficial).

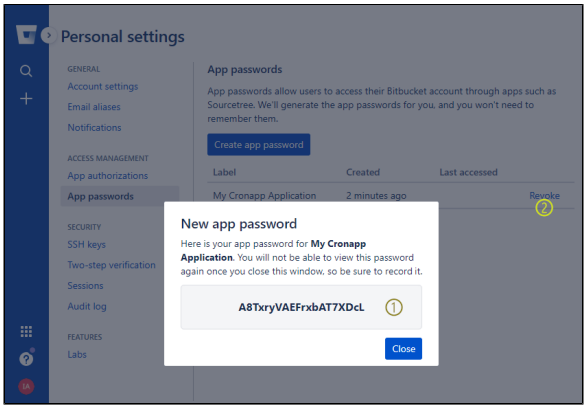

Figura 4.4 - *App password* gerado pelo Bitbucket

Caso queira excluir a senha, basta clicar em **Revoke** (destaque 2 da figura 4.4). Todos os projetos que possuam essa senha cadastrada perderão o acesso ao repositório.