Databricks
Padrão de Desenvolvimento

Histórico de versões

Versão	Data	Responsável	Descrição
1	09/2022	Ingrid Canaane – Ipiranga	Versão inicial do documento.
2	01/2023	Robson Quintino - Ipiranga	Ajustes no nome do ambiente produtivo e nos ambientes de sensíveis.
3	03/2023	Manoel Sá - BlueShift	Removidos menções/orientações ao Legado mantendo apenas orientações aos novos padrões, Ajustes nos tópicos: 2, 3.3, 4.1, 4.1.1, 4.2
4	04/2023	Manoel Sá - BlueShift	Novas orientações de versionamento, chamadas do criaListaCamadas e acessoCamadas, Workflows, Tabelas on-premises. Tópicos ajustados/inclusos: 2, 3.2, 3.4, 4.1.1, 5, 5.1, 5.2

Sumário

1. Introdução

Na Ipiranga, utilizamos o recurso Databricks com as linguagens ***PySpark*** e ***Spark SQL*** para manipular grandes conjuntos de dados, transformá-los, realizar limpeza nos dados, remoção de duplicidades, criação de novas tabelas, replicação de relatórios, seja na extensão CSV ou parquet, entre outras ações.

Dito isso, o objetivo deste documento é instruir os desenvolvedores na construção de processos utilizando o Databricks, apontando os principais pontos de atenção quanto à organização, padronização e metodologia de trabalho.

2. Ambientes

O ambiente de desenvolvimento agora encontra-se unificado, contendo apenas um Workspace Databricks denominado **dtb-ipp-dev**.

Neste ambiente, os desenvolvimentos serão concentrados dentro de "Repos", onde cada desenvolvedor poderá vincular o repositório da área a qual tem acesso dentro do seu usuário. (**Figura 11**)

Sendo assim, o desenvolvimento será feito no ambiente de DEV dentro da área correspondente e, após deploy, as alterações realizadas serão passadas para o workspace produtivo (a saber, ***dtb-ipp-prd*** via pipeline de CI/CD.)

No workspace de DEV, é possível fazer leitura e escrita no storage de laboratório, *stippdatalakelab*, utilizando o ponto de montagem ***/mnt/[área]/dev/***, e apenas leitura no storage produtivo, *stippdatalakedev*, utilizando o ponto de montagem ***/mnt/[área]/prd/***.

No workspace de **produção**, **só é possível fazer leitura e escrita no storage produtivo**, portanto, todo arquivo necessário para um processo deve estar disponível ali.

2.1 Dados sensíveis

Possuímos ainda um workspace Databricks, ***dtb-ipp-sensiveis-prd,*** utilizado para trabalhar com dados sensíveis, como dados cadastrais de nossos clientes, sejam dados bancários, CPF, endereço, telefone, email etc. Caso um projeto precise trabalhar com dados sensíveis, contidos no storage ***stippdatalakelgpdhml***, através do Databricks, deverá solicitar acesso ao ambiente de dados sensíveis, pois é o único que possui acesso de leitura e escrita para esse storage. O workspace ***dtb-ipp-sensiveis-dev*** é onde serão desenvolvidos e testados os processos que posteriormente passarão para o workspace produtivo, e possui acesso de leitura para o storage *stippdatalakelgpdhml*, e de leitura e escrita para o storage ***stippdatalakelgpdlab***.

3. Padrão de desenvolvimento

3.1 Organização dos notebooks

A organização dos notebooks dentro do workspace deve ser feita segundo o padrão abaixo:

[área]
- ***[projeto 1]***
  - ***config***
    - *acessoCamadas.py*
    - *criaListaCamadas.py*
  - ***[assunto 1]***
    - *[notebook 1]*
    - *[notebook 2]*
- ***[projeto 2]***
  - ***config***
    - *acessoCamadas.py*
    - *criaListaCamadas.py*
  - ***[assunto 1]***
    - *[notebook 1]*
  - ***[assunto 2]***
    - *[notebook 1]*

3.2 Pasta *config*

No repositório das áreas, cada projeto deverá ter a própria pasta ***config***. Dentro dela, deve haver 2 artefatos principais a serem usados em todos os notebooks, que são os arquivos ***criaListaCamadas.py*** e ***acessoCamadas.py***.

Nesses notebooks, haverá duas variáveis importantes. A primeira, ***escopoArea,*** é o nome da área a que pertence o workspace, e é usada para "construir" o ponto de montagem a ser usado nas funções de acesso a arquivos. A segunda, ***escopoProjeto,*** é o nome do projeto, e é usada para construir o caminho no sistema de arquivos do Databricks onde serão gravados os arquivos de apoio.

3.2.1 Notebook criaListaCamadas.py

Nesse notebook devem ser especificados os diretórios e arquivos do storage que serão acessados pelos processos.

São 3 chaves a serem preenchidas:

- ***baseLeitura***, onde devem estar os diretórios acessados para leitura;
- ***baseEscrita***, onde devem estar os diretórios acessados para escrita;
- ***baseArquivos***, onde devem estar os nomes dos arquivos que serão acessados.

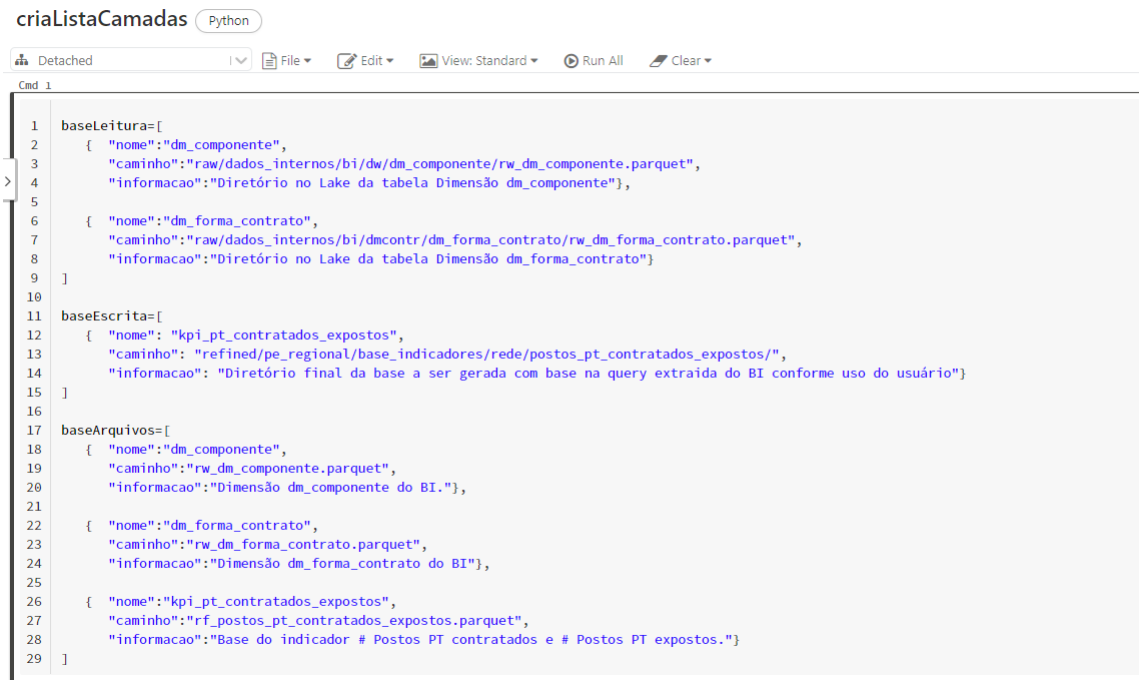

Figura 1: *Exemplo de preenchimento do notebook criaListaCamadas.py*

Se um diretório for acessado para leitura e escrita, deve ser especificado em ambas as chaves.

Dentro de cada uma dessas 3 chaves, conterão as informações sobre cada base que será usada, também agrupadas como chaves. A estrutura dessas sub-chaves deve conter:

- ***"nome":*** nome usado para referenciar aquela base, seja diretório ou nome de arquivo;
- ***"caminho":*** diretório em que se encontra o arquivo no Data Lake, seja este o diretório de leitura ou de escrita; no caso de ***baseArquivos***, deve ser preenchido com o nome do arquivo;
- ***"informacao":*** descrição da base em questão.

A partir das chaves principais, 3 arquivos json serão criados no sistema de arquivos: ***baseLeitura.json, baseEscrita.json*** *e* ***baseArquivos.json***. Eles serão usados no notebook *"acessoCamadas.py"*.

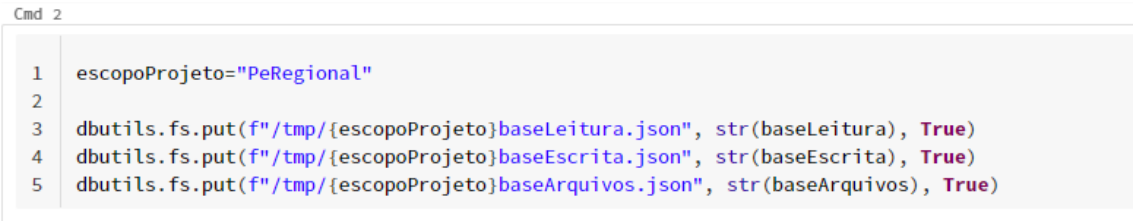

Figura 2: *Exemplo de preenchimento do notebook criaListaCamadas.py*

Importante: **o notebook "*criaListaCamadas*" deve ser executado no início de cada processo que o utiliza, para garantir que os arquivos de base estejam sempre atualizados.**

Com a utilização do "Repos" e para que em produção não ocorra "quebras" de chamadas dos notebooks criaListaCamadas assim como acessoCamadas (3.2.2), ambos devem ser chamados fazendo uso do caminho relativo, exemplo:

%run ../../config/criaListaCamadas

%run ../../config/acessoCamadas

%run ../config/criaListaCamadas

%run ../config/acessoCamadas

O caminho relativo vai mudar dependendo da hierarquia de pasta utilizada.

3.2.2 Notebook acessoCamadas.py

É um notebook padrão, que será disponibilizado na pasta do projeto na criação do ambiente. Define as variáveis e métodos a serem usados nos notebooks para acesso aos arquivos, para leitura e gravação. No notebook estão disponíveis instruções para uso, bem como alguns exemplos.

A partir dos arquivos base criados no notebook ***criaListaCamadas.py***, serão criados os *dataframes* usados nos métodos de leitura e escrita.

acessoCamadas Python

Detached | File | Edit | View: Standard | Run All | Clear

Cmd 1

```
from pyspark.sql.functions import col, lower
# Configuracao de camadas:

# Escopo do projeto
escopoArea='pco'
escopoProjeto="PeRegional"

# Ambiente DEV (Leitura e gravação)
mntDev="/mnt/" + escopoArea + "/dev/"

# Ambiente HML (somente leitura)
mntHml="/mnt/" + escopoArea + "/hml/"

# Ambiente PRD (somente leitura)
mntPrd="/mnt/" + escopoArea + "/prd/"

# Caminho para diretorio base
dfCaminhosLeitura = spark.read.option("multiline", "true").json(f"/tmp/{escopoProjeto}baseLeitura.json")
# baseLeituraExemplo1=mntPrd+dfCaminhos.filter("nome='sales_raw'").first().caminho

dfCaminhosEscrita = spark.read.option("multiline", "true").json(f"/tmp/{escopoProjeto}baseEscrita.json")
# baseEscritaExemplo2=mntDev+dfCaminhos.filter("nome='exemplo2'").first().caminho

dfArquivos = spark.read.option("multiline", "true").json(f"/tmp/{escopoProjeto}baseArquivos.json")

```

Figura 3: *Exemplo do notebook acessoCamadas.py*

Os mencionados métodos para acesso a arquivos irão retornar o diretório de um arquivo com base no nome da base e ambiente que forem passados como parâmetro.

O método ***caminhoLeitura*** buscará no arquivo *baseLeitura.json* o diretório correspondente ao nome de base informado. Caso seja passado também um valor para o parâmetro "*nomeArquivo*" (opcional), o nome informado em *baseArquivos.json* será concatenado com o diretório, retornando o caminho completo do arquivo.

Outro parâmetro obrigatório é o que se refere ao ambiente, que definirá se a leitura ou escrita será feita em ambiente produtivo (*ambienteDevHmlPrd = "prd"*) ou de laboratório (*ambienteDevHmlPrd = "dev"*). O retorno do método será a concatenação do ponto de montagem do ambiente informado via parâmetro com o restante do diretório. Dessa forma, se o diretório retornado não existir no ambiente desejado, ou esse diretório for usado para escrita em ambiente produtivo (a partir do ambiente de laboratório), a tentativa de leitura ou escrita retornará um erro, portanto, é necessário atenção durante o desenvolvimento.

```
def caminhoLeitura(ambienteDevHmlPrd="dev",nomeObjeto="", nomeArquivo=""):
  if nomeObjeto != "" and ambienteDevHmlPrd in ("dev","hml","prd"):
    if ambienteDevHmlPrd == "dev": ret=mntDev
    if ambienteDevHmlPrd == "hml": ret=mntHml
    if ambienteDevHmlPrd == "prd": ret=mntPrd
    ret=ret+dfCaminhosLeitura.filter("nome='"+nomeObjeto+"'").first().caminho
    if ret[-1:] !="/": ret=ret+"/"
    if nomeArquivo != "": ret=ret+dfCaminhosArquivos+dfNomes.filter("nome='"+nomeArquivo+"'").first().caminho
    return ret
```

Figura 4: método *caminhoLeitura*

O método ***caminhoEscrita***, funciona da mesma forma, porém, buscará os diretórios indicados no arquivo *baseEscrita.json*.

```
def caminhoEscrita(ambienteDevHmlPrd="dev",nomeObjeto="", nomeArquivo=""):
  if nomeObjeto != "" and ambienteDevHmlPrd in ("dev","hml","prd"):
    if ambienteDevHmlPrd == "dev": ret=mntDev
    if ambienteDevHmlPrd == "hml": ret=mntHml
    if ambienteDevHmlPrd == "prd": ret=mntPrd
    ret=ret+dfCaminhosEscrita.filter("nome='"+nomeObjeto+"'").first().caminho
    if ret[-1:] !="/": ret=ret+"/"
    if nomeArquivo != "": ret=ret+dfCaminhosArquivos+dfNomes.filter("nome='"+nomeArquivo+"'").first().caminho
    return ret
```

Figura 5: método *caminhoEscrita*

O método ***getPath*** tem funcionamento semelhante aos dois anteriores, porém serve para ambas as operações de leitura e escrita, sendo necessário informar um valor para o parâmetro *"tipo"*, que indicará qual será a operação ("E" para escrita, ou "L" para leitura). Isso determinará se o método recuperará o diretório no arquivo *baseEscrita.json* ou *baseLeitura.json*.

```python
def getPath(ambiente: str = "dev", nome: str = None, tipo: str = None, arquivo: str = None) -> str:
    path = None

    if nome is None or tipo is None:
        return path
    elif tipo.upper() in ("E", "L") and ambiente.upper() in ("DEV", "HML", "PRD"):
        if ambiente.upper() == "DEV": path = mntDev
        if ambiente.upper() == "HML": path = mntHml
        if ambiente.upper() == "PRD": path = mntPrd

    if tipo.upper() == "E":
        arquivoDF = spark.read.option("multiline", "true").json(f"/tmp/{escopoProjeto}baseEscrita.json")
    else:
        arquivoDF = spark.read.option("multiline", "true").json(f"/tmp/{escopoProjeto}baseLeitura.json")

    arquivoFilterDF = arquivoDF.filter(lower(col("nome")) == nome.lower())

    if arquivoFilterDF.count() > 0:
        path += arquivoFilterDF.first().caminho
    else:
        path = None

    if arquivo is not None:
        if path[-1:] != "/": path += "/"

        path += spark.read.option("multiline",
"true").json(f"/tmp/{escopoProjeto}baseArquivos.json").filter(lower(col("nome")) == arquivo.lower()).first().caminho

    return path
```

Figura 6: método *getPath*

3.3 Clusters

No modelo atual (Mencionado em 2. Ambientes) é importante e recomendado que ao desenvolver, selecione o cluster de sua área, esta visão de mais que um cluster é possível apenas para desenvolvedores que fazem parte e/ou tem acesso a outras áreas de desenvolvimentos, por isso é fortemente recomentado que selecione o correto.

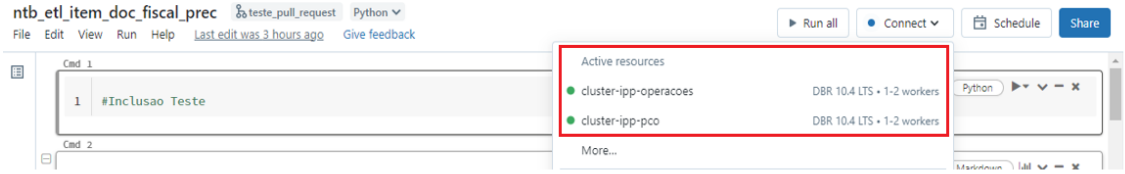

Figura 7: Listando Clusters disponíveis

No ambiente de DEV é disponibilizado um cluster para cada área, e seu nome segue o padrão ***cluster-[área]***. A versão da runtime usada é a última LTS (*Long Time Version*) disponível. Atualmente (setembro de 2022), é a 10.4 LTS, com Spark 3.2.1 e Scala 2.12.

Qualquer necessidade de atualização da versão do cluster deve ser alinhada com o time de Infra Cloud, para que seja verificada a viabilidade. Se a versão do cluster usada durante o desenvolvimento não corresponder a do ambiente produtivo, é possível que ocorra erro na execução após o deploy. A situação inversa também é verdadeira, uma vez que alterar a versão do cluster produtivo pode impactar no funcionamento de processos que já estão ali.

3.4 Workflows

Não orientamos a criação de *Workflows* nos ambientes de desenvolvimentos, entende-se que se o processo tem necessidade de execução com uma frequência e está vinculado a uma demanda existente de uso, o mesmo deverá passar pela esteira de produtização para que possa ser agendado via Data Factory.

Existe uma rotina diária que faz a deleção de todos os *Workflows* que possivelmente foram criados, esta rotina não impacta em nenhum notebook desenvolvido, apenas deleta todos os *Workflows* seja ele em execução ou inativo.

3.5 Boas práticas e orientações gerais

- Usar o ***PySpark*** no lugar do Python puro, uma vez que este inviabiliza o uso do processamento paralelo pelo Spark;
- Caso necessário utilizar a biblioteca pandas, importar a do Spark, ***pyspark.pandas***, uma vez que o Spark é multithread e seu código pode ser executado de forma distribuída;
- Sempre dar preferência à utilização de bibliotecas que são *built in* do Spark. Necessidades que fujam dessa orientação devem ser levadas ao time de Governança Técnica, que avaliará a possibilidade de instalação;
- Não fazer leitura de arquivos *XLSX* ou *XLS* no Databricks, uma vez que esta extensão reduz consideravelmente o desempenho do cluster. Para trabalhar com dados que originalmente possuem esse tipo de extensão, recomendados a utilização do Data Factory para conversão para parquet;
- Não fazer leitura de um diretório completo, a menos que seja indispensável. Caso a necessidade do processo seja ler um range de tempo específico, montar o diretório de forma a ler apenas os anos, meses ou dias desejados. Por exemplo, se uma tabela tem histórico de 2013 a 2022 no Data Lake, mas um processo que a usará precisa apenas dos dados de 2021 em diante, deve-se ler apenas esse período de tempo, evitando, assim, uma leitura desnecessária.
- Não é permitido produtizar processos apontando para o workspace de DEV, portanto, toda alteração deve ser produtizada, ou os outputs corretos serão escritos apenas no storage de LAB, cuja utilização não será liberada em outros processos;
- Arquivos temporários de processos devem ser escritos na camada transient, conforme padrão de camadas definido, e excluídos ao final do processo;
- Ao realizar escrita no storage, o Databricks gera arquivos temporários e com nomes fora do padrão usado pela Ipiranga. Para solucionar isso, usamos um método simples que faz a escrita inicial na camada transient, e depois faz cópia apenas do arquivo parquet para a camada final (trusted ou refined), renomeando conforme padrão.

```
# Grava arquivo no Lake, Cria Cópia [Origem >> Destino] renomeando o arquivo,
# ao final remove da camada transient

def copia_arquivo(path_escrita_rf,path_escrita_tt,df,nome_arquivo):
  # Grava Arquivo na Camada TT
  (df.coalesce(1)
        .write
        .format("parquet")
        .mode("overwrite")
        .save(path_escrita_tt,header=True)
        )
  # Lista Arquivo
  arquivo = dbutils.fs.ls(path_escrita_tt)[-1][0]

  # Realiza Cópia da camada TT para a camada RF atribuindo novo nome
  dbutils.fs.cp(arquivo,path_escrita_rf+nome_arquivo)

  # Remove os dados gravados na TT
  dbutils.fs.rm(path_escrita_tt,True)
```

Figura 8: método *copia_arquivo*

3.6 Uso de Delta Table

O uso de Delta Tables é indicado quando há necessidade de controle de versão de registros para o output de um processo, ou seja, quando é realizado operações de *insert* e/ou *update* de registros.

Se o processo contempla, por exemplo, reprocessamento full dos dados ou de um período de tempo, ou processa sempre dados de uma data de referência, a orientação é gravá-los como parquets padrão, segundo o padrão de camadas em Definição de Camadas do Data Lake.

4. Fluxo de produtização

4.1 Versionamento

Será demonstrado a forma mais atual de versionamento dos notebooks no DevOps para posterior produtização, que é através do uso da funcionalidade “Repos” do Databricks.

4.1.1 Utilizando a funcionalidade Repos

No DevOps do projeto, o engenheiro deverá criar uma branch no repositório do Databricks, cujo nome segue o padrão ***prj-[área]-adb***, a partir da branch principal (main/master), conforme imagem abaixo:

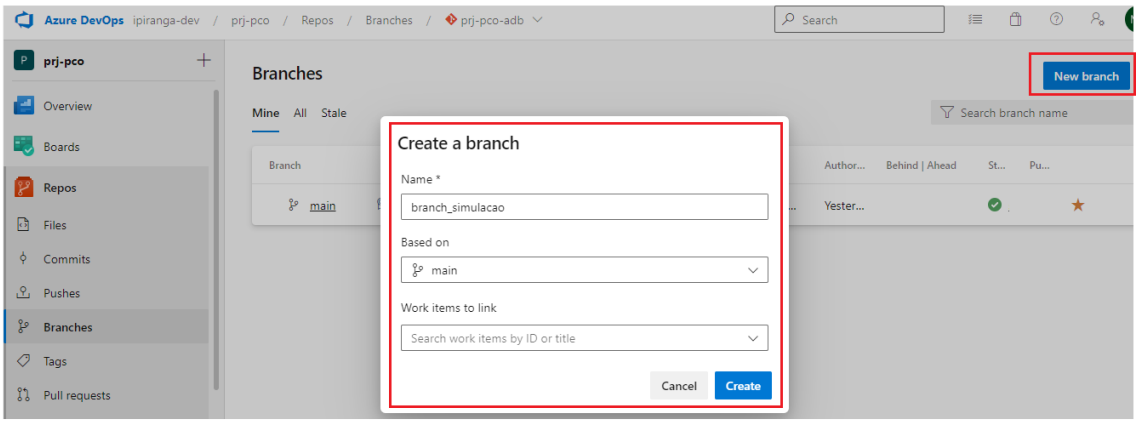

Figura 9: Criando Branch no repositório

Após branch criada, é hora de obter o link para realizar o clone do repositório diretamente no Databricks.

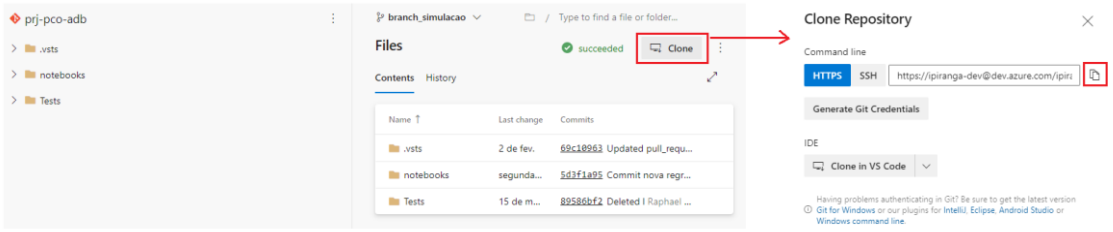

Figura 10: Obtendo link para clone

No ambiente de desenvolvimento, dentro de ***Repos > pasta usuário***, adicionar o repositório o qual foi feito o clone anteriormente.

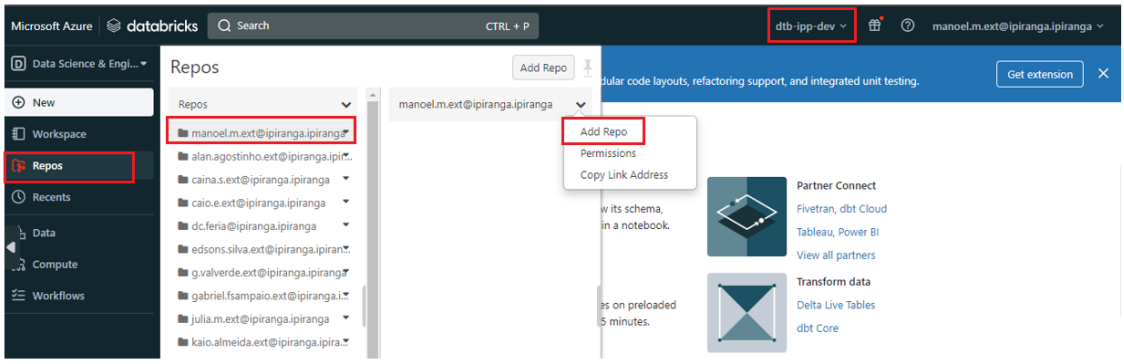

Figura 11: Adicionando repositório clonado

Após colar o link copiado, as demais informações como *Git provider* e *Repository name* são preenchidos de forma automática, basta seguir para a criação do repositório.

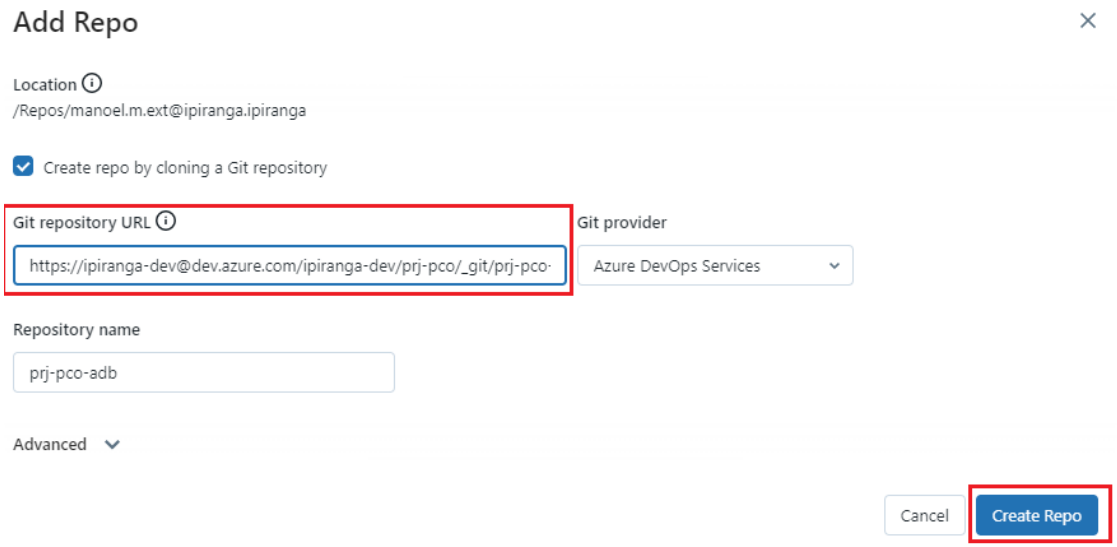

Figura 12: Finalizando criação de repositório

Seu ambiente terá algo como:

Figura 13: Resultado pós criação do repositório

Para selecionar a branch criada anteriormente, basta seguir os passos destacados, uma nova janela será mostrada a qual possibilitará escolher a branch desejada.

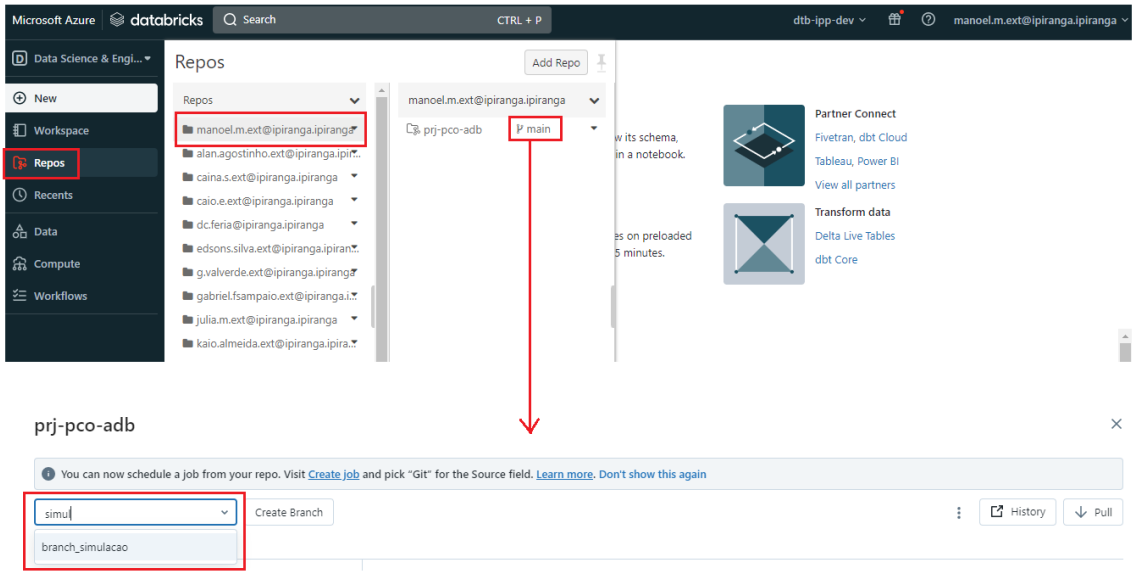

Figura 14: Selecionando branch

Obs.: Este processo de clonar o repositório é necessário apenas uma vez.

Para criar novas branchs de desenvolvimento, basta seguir os passos:

- Dentro do seu usuário e no repositório desejado, ao clicar em branch uma nova aba será exibida, certifique-se de que selecionou a **branch principal (main/master)**, aplique um **pull** para sincronizar os ativos, após isso é só criar sua nova branch.

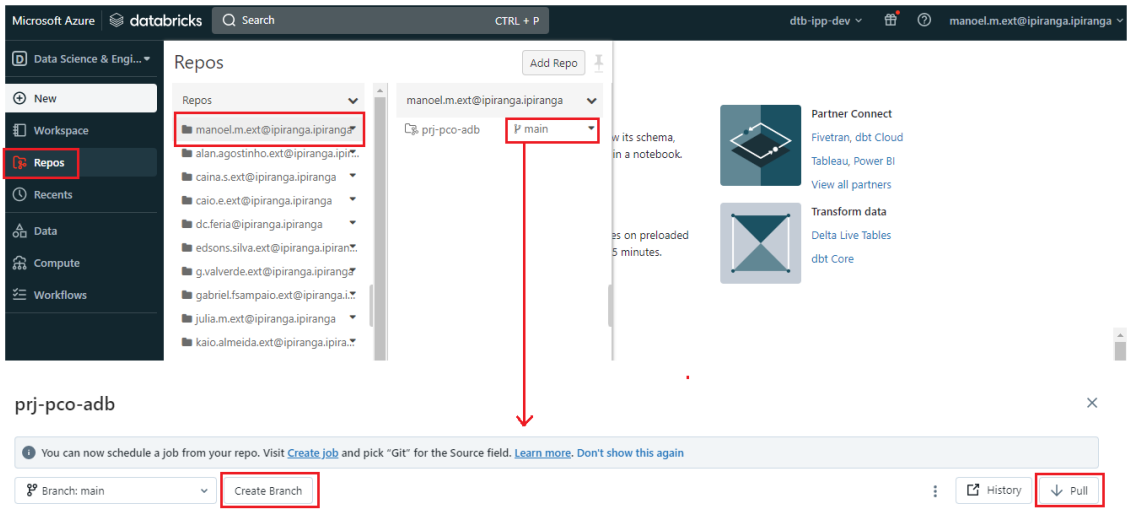

Figura 15: Criando nova branch a partir da principal (main/master)

É possível verificar os commits realizados na branch através do DevOps.

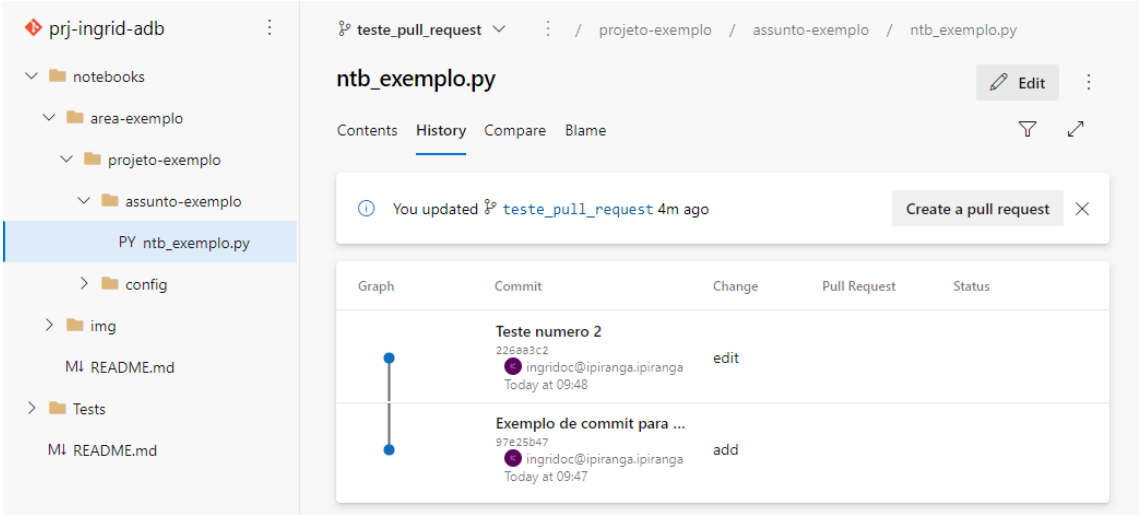

Figura 16: *Exemplo 2 de tela de histórico de commits de um notebook no repositório*

Caso necessário, é possível recuperar uma versão anterior de um notebook através da aba de ***Last edit***. Nesse exemplo, apagamos uma célula que já havia sido commitada (Figura 17). Navegando pela aba lateral direita, podemos visualizar as versões anteriores do notebook, e para restaurar a versão desejada, basta clicar sobre ela e então na opção ***Restore this revision*** (Figura 18). Aparecerá uma mensagem para confirmação e, feito isso, o notebook estará de volta ao seu estado anterior (essa ação conta como uma alteração, portanto, é necessário commitar novamente).

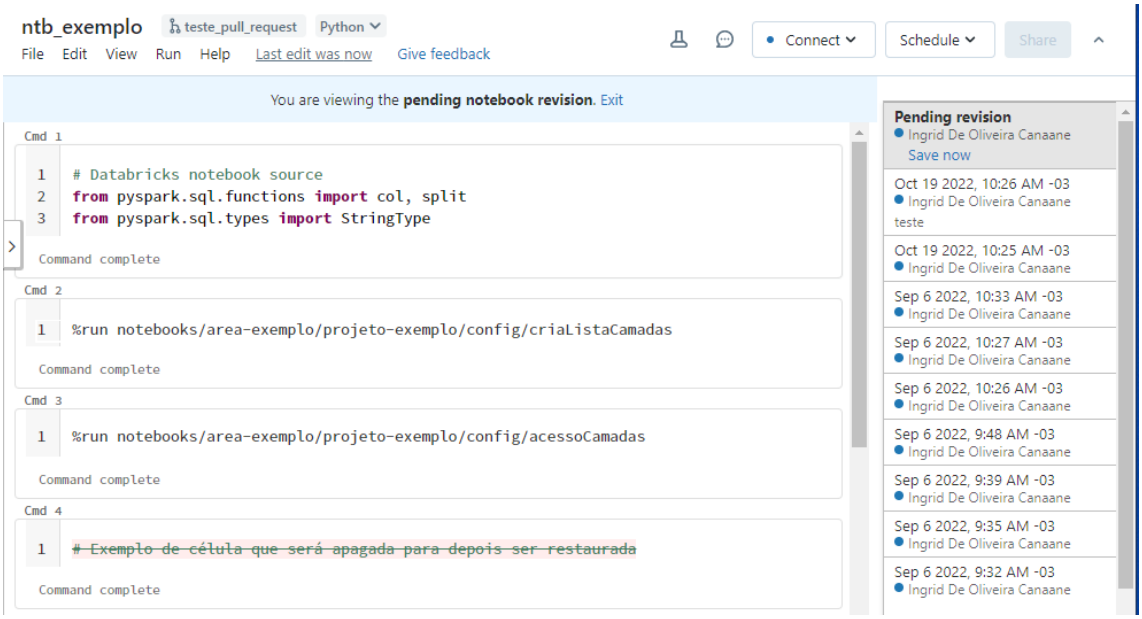

Figura 17: Exemplo de restauração de notebook a uma versão anterior, parte 1

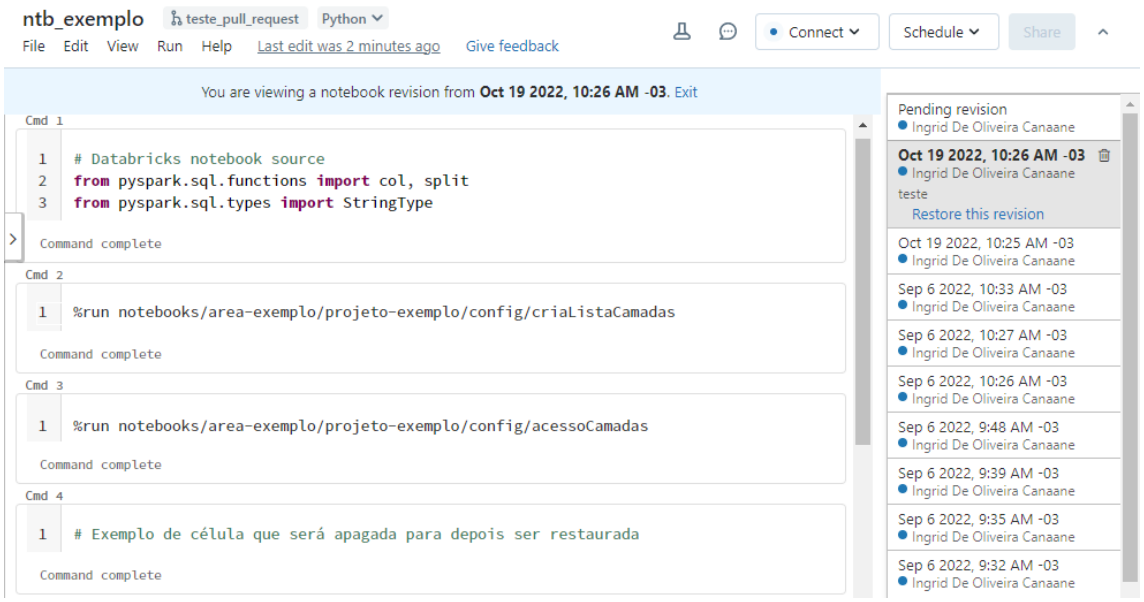

Figura 18: Exemplo de restauração de notebook a uma versão anterior, parte 2

4.2 Criação de Pull Request

Após a conclusão dos desenvolvimentos e realizado o processo de versionamento para cada notebook alterado/criado, é hora de criar o Pull Request, então neste caso, voltar para sua branch (itens 1, 2 e 3 **Figura 19**) ou simplesmente clicar na branch a qual é mostrada no notebook do desenvolvimento (**Figura 20**) e seguir os demais passos:

4. Incluir mensagem do commit (requerido) assim como uma breve descrição (opcional)
5. Realizar o **commit & Push** das mudanças, caso deseje realizar o commit para apenas alguns processos, basta desmarcar as caixas a esquerda onde mostra como "**changed file**" no exemplo temos apenas 1 arquivo em evidencia. A alteração escolhida será commitada, e as demais permanecerão pendentes. Uma mensagem de sucesso deve aparecer nessa mesma tela.
6. Após o commit, é possível visualizar um link para criar um pull request, este link é apenas um facilitador para direcionar ao repositório diretamente a aba Pull Requests.
7. Nesta etapa de criação de Pull Request basta seguir o preenchimento do **Checklist** o qual será exibido na tela, importante preencher/conferir cada tópico do **checklist**, pois são parâmetros considerados durante as avaliações, por fim clicar em "**Create**" para concluir o processo.

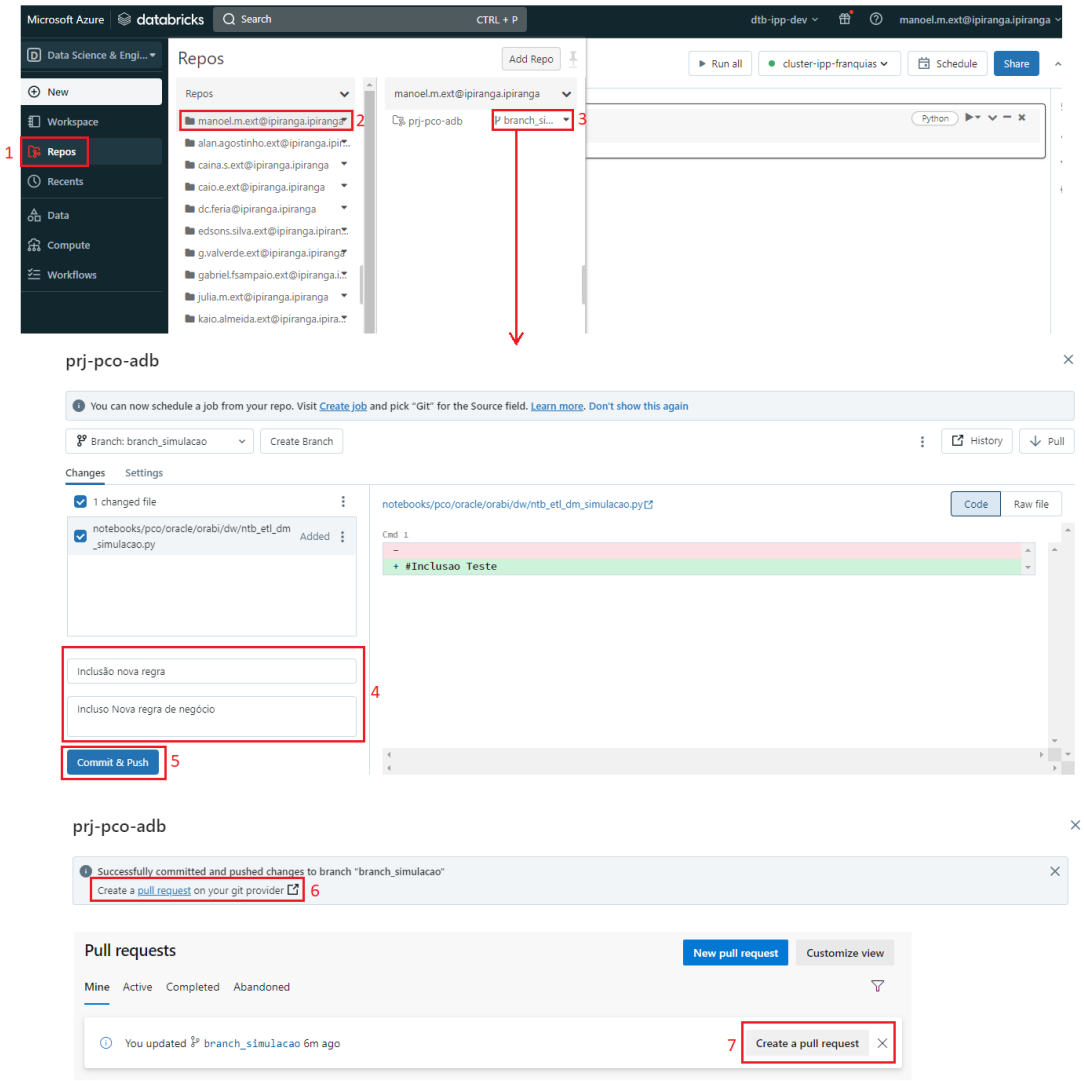

Figura 19: Criando Pull Requets

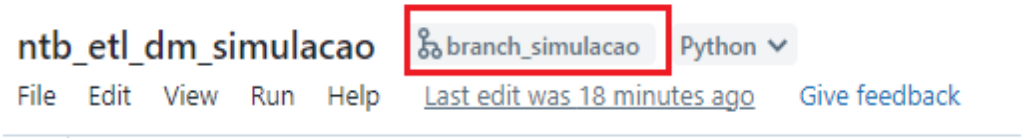

Figura 20: Selecionando branch para criar pull request.

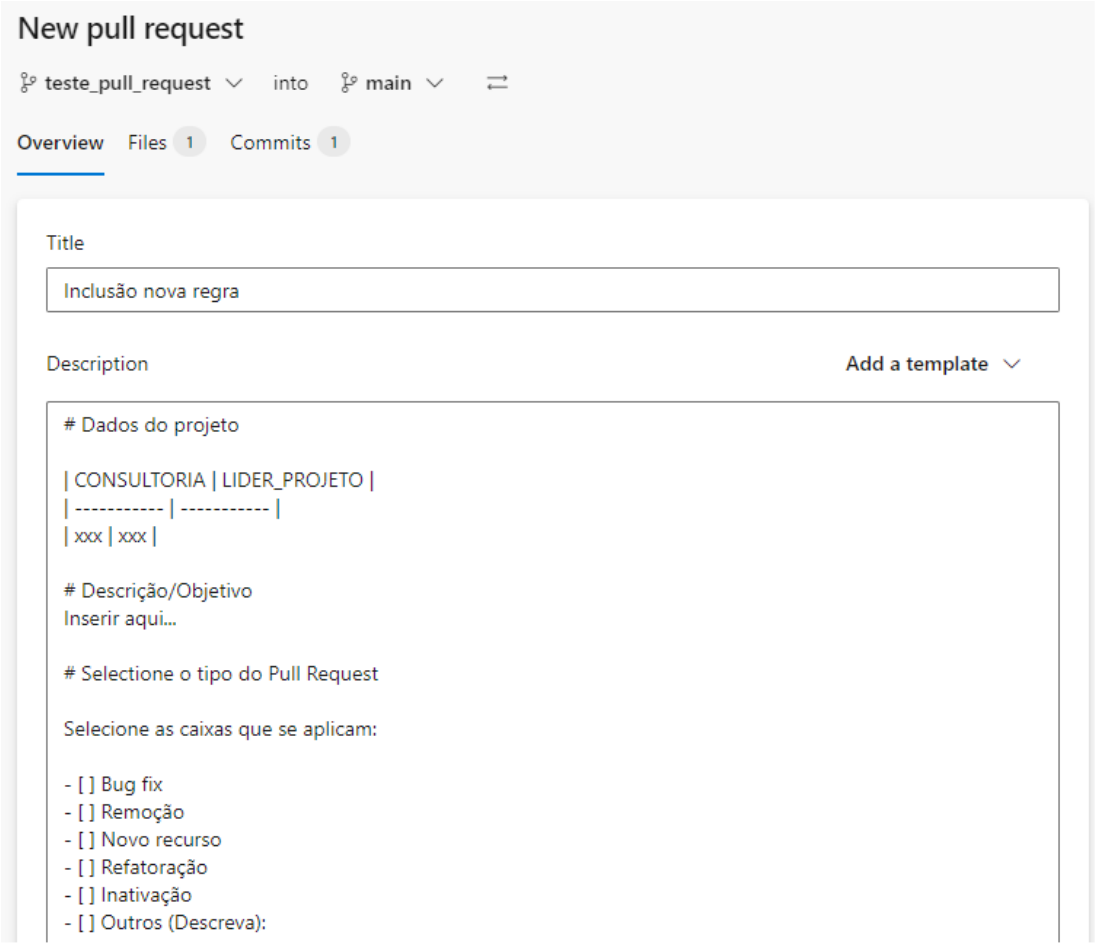

Figura 21: Preenchendo checklist antes do "**Create**"

Preenchidas as informações do Checklist contendo as evidencias necessárias/requisitadas e criado o PR, este cairá para avaliação do time de Governança Técnica, que fará análise e aprovação das alterações.

Após a aprovação do PR, o pipeline de CI/CD fará a passagem das alterações para o ambiente produtivo, workspace **dtb-ipp-prd**.

5. Tabelas on-premises

Em alguns casos é necessário realizar tratamentos em tabelas após a ingestão para que seus dados fiquem confiáveis como por exemplo sem duplicidades em seus registros, podendo o dado ficar armazenado na camada trusted, é comum o uso do Databricks para realizar esta operação, porém com o intuito de concentrar tabelas desta

natureza em um único local e termos uma melhor organização assim como facilidade em suas manutenções, será demonstrado nos próximos passos onde e como armazenar estes desenvolvimentos.

Obs.: Estes processos de ingestões são referentes as fontes: ORACLE, ABADI e JDE

5.1 Ambiente de desenvolvimento

Os notebooks devem ser desenvolvidos dentro de um dos Workspace/repositórios abaixo dependendo da natureza do dado

- **dtb-ipp-dev (para dados que não possuem características sensíveis)**
  - prj-pco >> prj-pco-adb
- **dtb-ipp-sensiveis-dev (para dados que possuem características sensíveis)**
  - prj-pco >> prj-pco-sensiveis-adb

5.2 Hierarquia de pastas

Será mantido uma hierarquia similar ao que hoje existe no Data Factory

Figura 22: Exemplo de hierarquia