

MANUAL DE CONCEITOS, NOMENCLATURAS E PADRÕES DE DESENVOLVIMENTO

- DATA LAKE INSTITUCIONAL-

Este manual apresenta os principais conceitos relativos a big data, nomenclaturas padrão a serem seguidos em iniciativas realizadas no âmbito do data lake institucional além dos padrões de desenvolvimento estabelecidos para ingestão e tratamento de dados. Trata-se de um documento norteador cuja revisão é de responsabilidade da Coordenação de Inteligência da Informação. Todos os padrões aqui apresentados são de uso obrigatório pelos fornecedores contratados para atuar em soluções de dados no ambiente cloud da instituição.

Coordenação de Inteligência da Informação
Gerência de Soluções do Negócio

Sumário

1. Conceitos.....	3
1.1 Arquitetura do Enterprise Data Hub	4
1.2 Arquitetura Técnica do EDH	5
1.3 Camadas de dados no Data Lake	5
1.4 Principais componentes da plataforma de Big Data	6
1.5 Links úteis.....	8
1.6 Pré-requisitos de acesso para os Desenvolvedores:	8
1.7 Pré-requisitos de acesso ao DevOps STI-CNI	10
2. Padrões de Nomenclatura.....	11
2.1 Geral	11
2.2 Data Lake Storage / Enterprise Data Hub (EDH)	11
2.3 Campos.....	13
2.3.1 Natureza	13
2.3.2 Substantivos, Qualificadores e Abreviaturas	14
2.3.4 Campos de controle	15
2.4 Valores padrão para métricas órfãs	15
2.5 Processos de carga no Data Lake Storage	16
3. Sistemas internos - Padrões de desenvolvimento – Ingestão de dados na camada RAW.....	17
3.1 Fontes OLTP	17
3.1.1 Pré-requisitos para os Desenvolvedores.....	17
3.1.2 Pré-requisitos CNI-STI	17
3.1.3 Desenvolvimento	18
3.1.4 Testes de execução.....	26
3.1.5 Publicação em produção.....	27
3.1.6 Camada RAW	27
4. Crawler / Bots (qualquer programa que busca dados externos) - Padrões de desenvolvimento – Ingestão de dados na camada RAW	28
4.1 Configuração do ambiente local no cofre de senhas	28
4.2 Desenvolvimento	28
4.3 Testes de execução.....	32
4.4 Resumo de crawlers	32
4.5 Publicação em DEV.....	33
4.6 Publicação em produção	35
4.8 Desenvolvimento Azure Data Factory	38
5. Camada RAW.....	41

5.1 – Camada RAW – Desenvolvimento com Azure Databricks.....	44
5.2 – Camada RAW – Transformações com Azure Databricks	45
5.2.1– Exemplo de uso de arquivos de parametrização	47
5.3 – Camada RAW – Orquestração com Azure Data Factory.....	47
6. Padrões de desenvolvimento – camada TRS.....	51
6.1 Desenvolvimento no Azure Databricks	51
6.2 Desenvolvimento no Azure Data Factory	57
7. Padrões de desenvolvimento – camada BIZ	59
7.1 – Desenvolvimento no Azure Databricks	59
7.2 Desenvolvimento no Azure Data Factory	63
8. Padrões de desenvolvimento - Workflows.....	64
8.1 Desenvolvimento dos Workflows no Azure Data Factory	64
9. Controle de versão	66

1. Conceitos

Uma arquitetura de Big Data é projetada para lidar com processamento e análise de grandes volumes de dados cujos processos são complexos demais para serem tratados em sistemas de bancos de dados tradicionais.

A **Figura 1** apresenta os componentes técnicos que se inserem na arquitetura de Big Data das ENSI, cujo ambiente computacional é provido pela Microsoft por meio da nuvem Azure.

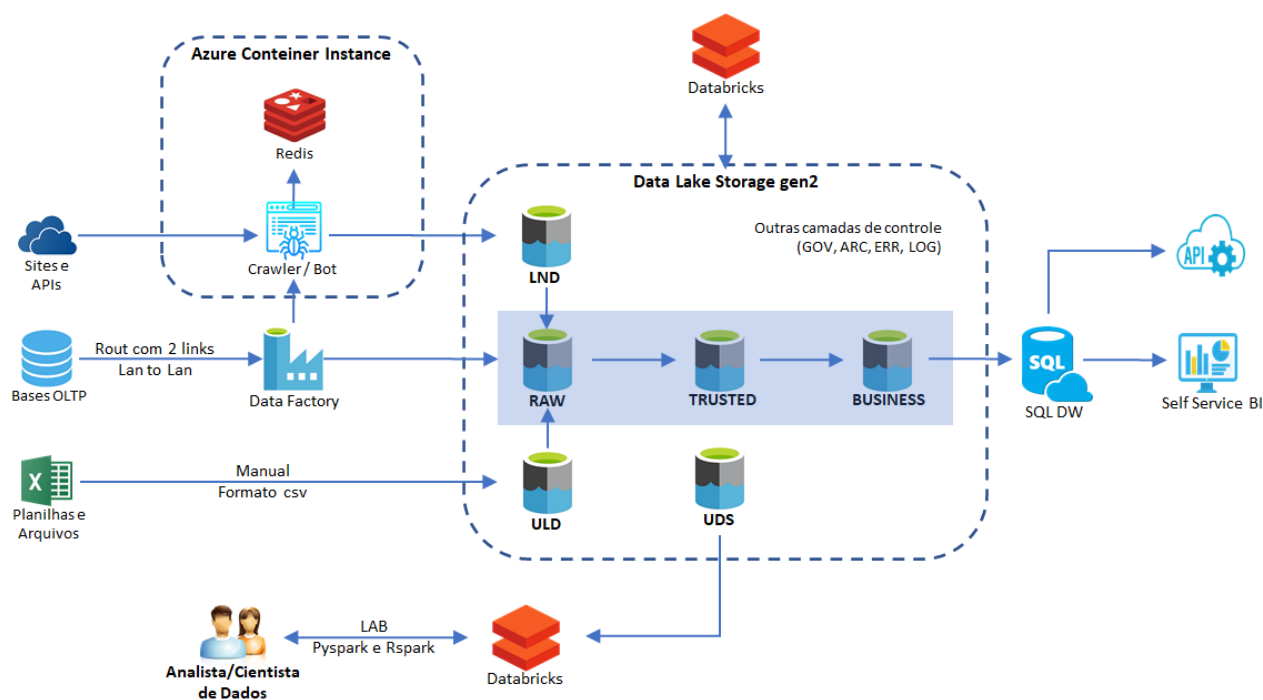


Figura 1 - Arquitetura Técnica (componentes utilizados)

1.1 Arquitetura do Enterprise Data Hub

O objetivo de um Enterprise Data Hub (EDH) é permitir que a instituição tenha uma fonte de dados centralizada e unificada que possa fornecer rapidamente informações a diversos usuários de negócio, apoiando a tomada de decisão.

O Enterprise Data Hub inclui:

Reservatório de Dados (Data Lake): Coleta de dados brutos que antes tinham alto custo para armazenamento e processamento. Os dados de diferentes fontes são gerenciados e governados no Data Lake, que também pode atuar como um arquivo online para dados acessados com menos frequência.

Refino de dados: Otimização do processo de integração de diversos tipos de dados de várias fontes para descobrir as relações. Analisar, limpar, transformar e integrar dados.

Exploração do Big Data: Realizar análises investigativas em grandes volumes de dados, aplicando técnicas de machine learning, de estatística e de análise de dados para descobrir novos conhecimentos.

Fácil acesso aos dados: Os mais variados tipos de dados podem ser facilmente acessados em um EDH, garantindo uma fonte única para o trabalho de analytics.

Armazenamento de dados em formato nativo: Uma das principais vantagens do EDH. Ao iniciar o trabalho de analytics, há a garantia que os dados estão em seu estado bruto, evitando distorção no processo de análise.

1.2 Arquitetura Técnica do EDH

A imagem a seguir apresenta a arquitetura técnica do EDH:

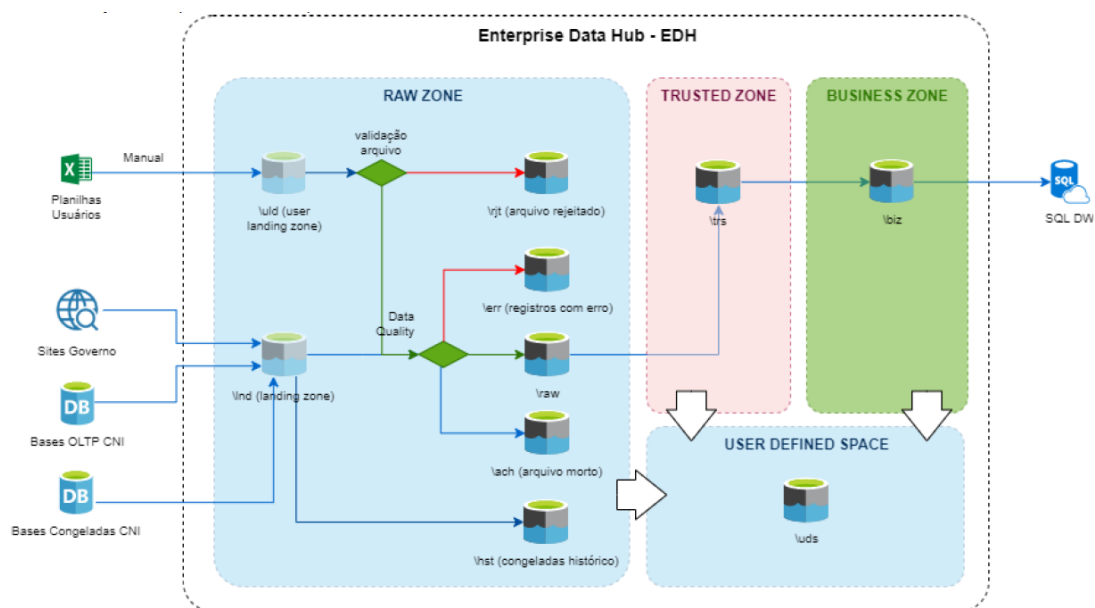


Figura 2 - Arquitetura Técnica EDH

1.3 Camadas de dados no Data Lake

Os produtos de desenvolvimento de código fonte entregues deverão disponibilizar os dados na plataforma Big Data em 3 camadas:

- **Dados Brutos (Raw):** cópia dos dados no mesmo formato da origem (*as is*);
- **Dados confiáveis (Trusted):** dados padronizados, atendendo às regras de negócio definidas pelos usuários;
- **Dados transformados (Business):** dados modelados e otimizados para melhor performance de consumo da área usuária.

1.4 Principais componentes da plataforma de Big Data

- **Portal Azure**: é o local unificado para a gestão dos recursos da nuvem Azure na assinatura do Big Data-CNI pelos administradores da conta.

- **Redis**: é um banco de dados NoSQL com suporte a vários tipos de estrutura de dados e muito rápido no retorno das requisições. É comumente utilizado para cache e gravação de estado de objetos. No projeto é utilizado de duas formas:

1. Nos Crawlers: banco chave-valor para guardar os status de coleta das fontes externas e permitir a retomada da coleta a partir do ponto de parada. Essa questão é crucial para APIs que fazem paginação, como Facebook e Youtube, por exemplo.
2. ADLS ACL Reporter: Cache para os dados dos usuários e grupos do AAD.

- **Data Factory (ADF)**: é um serviço de ETL de integração de dados baseado em nuvem.

No projeto ele é utilizado para executar os fluxos de automatização de cargas, desde ingestão dos dados, passando pela movimentação dos arquivos no DataLake, executando os scripts criados na esteira de produção via DataBricks para a transformação dos dados e gravando os dados do SQLDW.

- **SHIR (Self Hosted Integration Runtime)**: é a infraestrutura de computação usada pelo Azure Data Factory para fornecer funcionalidades de integração de dados entre diferentes ambientes de rede.

No projeto ele é utilizado em uma VM, conforme orientações da Microsoft, para realizar a ponte entre as bases de dados dos sistemas (OLTP - *Online Transaction Processing*) e a nuvem, ou seja, é uma espécie de servidor confiável que tem autorização de buscar os dados dos sistemas e disponibilizá-los na nuvem (via DataFactory) de forma segura.

- **Container**: é uma tecnologia que padroniza, empacota e torna portátil uma determinada aplicação. Ele agrupa o código de um aplicativo, seus respectivos arquivos de configuração e as bibliotecas necessárias para a sua execução.

No projeto os containers são utilizados para o desenvolvimento e a execução dos crawlers que buscam dados de sites externos (arquivos, raspagens, APIs, outros) para o Data Lake.

- **ADLS (Azure Data Lake Storage Gen2):** é um repositório escalável para cargas de trabalho analíticas de big data. O Azure Data Lake permite a captura de dados de qualquer tamanho, tipo e velocidade de ingestão em um único lugar para análises operacionais e exploratórias.

No projeto ele é utilizado como o repositório oficial dos dados de toda a instituição, cujos arquivos podem ser acessados via DataBricks e DataFactory.

- **DataBricks:** é um componente para análise de dados baseado no Apache Spark. Ele possibilita o processamento dos dados em memória distribuída, tornando rápidas as soluções de cálculos complexos.

Na arquitetura da CNI o DataBricks é utilizado como serviço de transformação de dados e de laboratório de ciência de dados. O PySpark é uma biblioteca que viabiliza o uso de Spark por meio da linguagem Python.

Através do PySpark é possível utilizar a sintaxe da linguagem Python em funções preparadas para realizar processamento de dados (junções, transformações, leitura, gravação) de modo distribuído (em cluster).

No DataBricks, há também a possibilidade de utilização das linguagens SCALA, R e SQL. No entanto, o Python (por meio da biblioteca PySpark) foi a linguagem definida como padrão para tratamento de dados na arquitetura da CNI devido a sua ampla gama de funções preparadas para a transformação de dados, grande volume de exemplos documentados, alta aderência da comunidade de desenvolvedores e menor curva de aprendizado.

- **SqIDW (atualmente chamado de Azure Synapse Analytics):** O Synapse é um sistema de data warehouse empresarial baseado em nuvem que aproveita o processamento paralelo maciço para executar rapidamente consultas complexas em petabytes de dados.

No projeto ele é utilizado para concentrar os conjuntos de dados agregados para a utilização no Tableau (ferramenta institucional de Self Service BI).

- **Tableau:** é uma plataforma integrada de Self Service BI que permite ao próprio usuário de negócio, realizar por conta própria, a criação de painéis de análise e visualização de dados.

No projeto ele é utilizado exatamente conforme sua definição. Trata-se da ferramenta de BI institucional e possibilita ao próprio usuário da área de negócio realizar suas análises e divulgar seu trabalho.

▪ **Functions**: O Azure Functions é um serviço de computação sem servidor que permite a execução de scripts de código disparado por meio de eventos e sem a necessidade de provisionar explicitamente ou gerenciar a infraestrutura.

No projeto as funções são utilizadas para integrar alguns componentes (auxiliar no cadastro de metadados, iniciar e finalizar as instâncias de containers, iniciar e finalizar os clusters) e realizar chamadas de serviços tais como crawlers e proxy.

1.5 Links úteis

Alguns os links de acesso direto aos recursos com o seu usuário do Azure AD:

- Databricks -> <https://eastus2.azuredatabricks.net/>
- Data Factory -> <https://adf.azure.com/home?factory=%2Fsubscriptions%2F3e3a21e3-9551-4945-8c46-c02f558392ce%2FresourceGroups%2Fbigdata%2Fproviders%2FMicrosoft.DataFactory%2Ffactories%2Fcni-bigdatafactory>
- DevOps -> <https://dev.azure.com/CNI-STI/> (projeto ENSI-BIG DATA)

1.6 Pré-requisitos de acesso para os Desenvolvedores:

Primeiramente, se o desenvolvedor é um usuário externo ao domínio da CNI, deve ser criado um usuário terceirizado no AD para que o acesso seja concedido.

Após a criação do usuário, ele deve ser vinculado a um grupo no AD e em seguida sincronizado ao Azure AD (AAD). Adotamos o seguinte padrão de nomenclatura para a criação de grupos no AD, no caso do usuário ser de um novo fornecedor, o padrão é GB-ASC-ACTI-<NOME EMPRESA>. Exemplos: GB-ASC-ACTI-QSOFT, GB-ASC-ACTI-OSBR, GB-ASC-ACTI-SenaiCE.

Acesso à VPN da CNI e Acesso ao cofre de senhas

Após a criação dos usuários e grupos no AD/AAD esses usuários devem ter o acesso ao cofre de senhas (máquina monitorada, segura e preparada para o desenvolvimento) via VPN (toda a orientação do primeiro acesso nesse ambiente é repassada pela equipe de infraestrutura), nessa máquina o desenvolvedor terá todos os softwares necessários para o desenvolvimento da ingestão de dados via cralwers/bot e também para as demais tarefas no âmbito da CNI, a seguir a lista de softwares instalados:

- Python3.6+- Redis 2.4.5 localhost (orientações de start do serviço disponíveis no caminho "C:/tmp/txt_env")
- PyCharm ou Virtual Studio Code
- Teams
- Navegadores Chrome e Firefox para acessar o portal da Azure e o projeto no devops sem nenhum risco de bloqueio de firewall
- Azure Storage Explore

Observação: no mesmo caminho que as orientações para start do Redis em localhost estão disponíveis ("C:/tmp/txt_env") também estão disponíveis as variáveis de ambiente que o desenvolvedor deve criar antes de executar os projetos no Pycharm ou no VS Code.

Por fim, com o usuário e grupos no AD/AAD criados, acesso via VPN ao cofre de senhas concedido, testado e o fornecedor tenha configurado e seu ambiente (solicitações atendidas normalmente pela fila ADM Windows), basta o grupo no AD/AAD criado receba as devidas permissões nos recursos da Azure utilizados no projeto, então na subscrição do projeto ensi-big data o grupo do fornecedor externo (GB-ASC-ACTI-<NOME EMPRESA>) deve possuir as seguintes permissões (essas permissões são concedidas pela fila ADM Cloud):

- 1 - Permissão de leitura nos recursos cnibigdatabricks1 e cnibigdatadlsgen2.
- 2 - Permissão de colaborador no recurso cnibigdatafactory.
- 3 – Acesso ao projeto ENSI-BIG DATA no DevOps CNI-STI com o perfil "Basic" para poder contribuir no projeto.
- 4 – Por fim, no projeto ENSI-BIG DATA no DevOps o grupo no AD dos fornecedores deve ser associado ao grupo "acesso de terceiros" nas permissões do projeto.

1.7 Pré-requisitos de acesso ao DevOps STI-CNI

- Vincular o desenvolvedor a um grupo com o prefixo “ANALISTA_” no AD/AAD. Exemplo de grupos já existentes: ANALISTA_STI, ANALISTA_UNIEPRO, ANALISTA_UNIGEST
- Adicionar o usuário ao *Azure DevOps* com *Access Level = Basic*

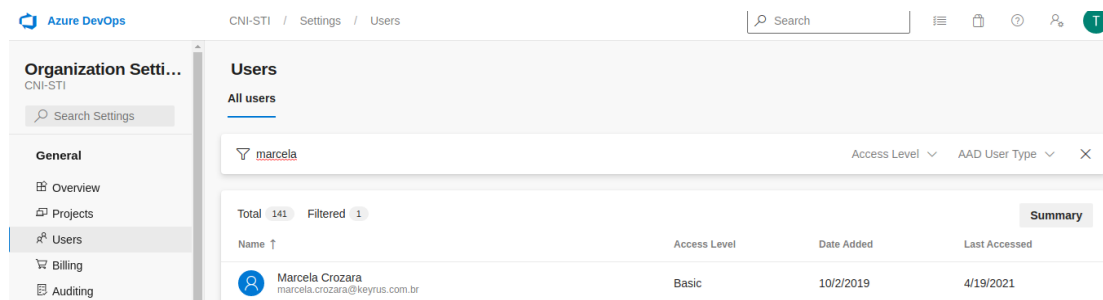


Figura 22

- Adicionar o grupo/usuário ao repositório *ENSI-BIGDATA CRAWLER* no *Azure DevOps* e garantir a esse grupo/usuário no máximo as permissões definidas para o grupo *Contributors* deste repositório ou, se for um usuário da equipe interna, basta adicioná-lo ao grupo do AD/AAD *DESENV_SUSTENTACAO_STI*.

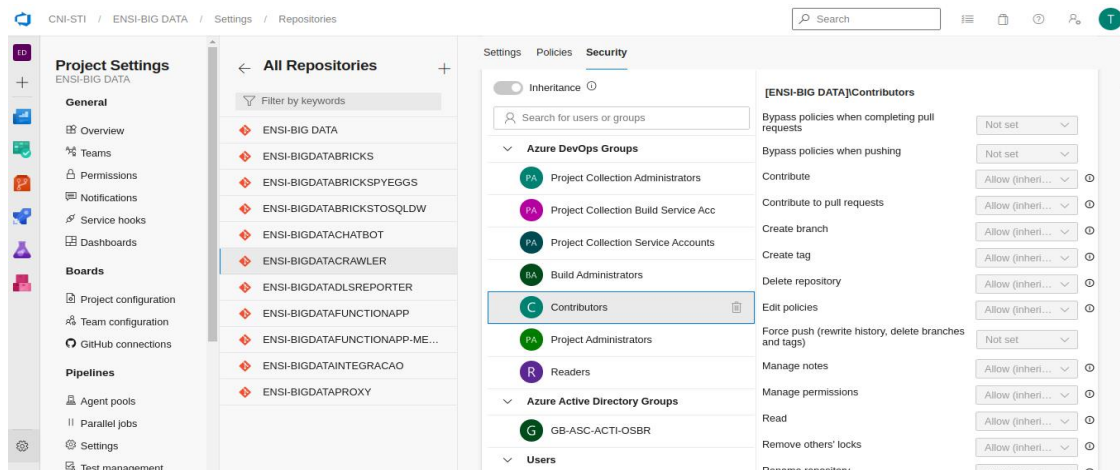


Figura 23

2. Padrões de Nomenclatura

Em um contexto de big data é possível observar a tendência de ingestão contínua de bases e desenvolvimento de análises de negócio em um reservatório central de dados: um data lake institucional. Para imprimir diretrizes básicas de organização das informações contidas nessa arquitetura é importante definir uma lógica simples de armazenamento, tratamento e categorização do conteúdo. Para isto, foram definidos os padrões de nomenclatura do data lake da CNI.

2.1 Geral

Regras gerais:

- Nomes dos objetos sempre em minúsculas com palavras separadas por "_";
- Nomes dos objetos com no máximo 5 termos descritos evitando abreviação;
- Não utilizar caracteres especiais (acento, cedilha, símbolos etc).

2.2 Data Lake Storage / Enterprise Data Hub (EDH)

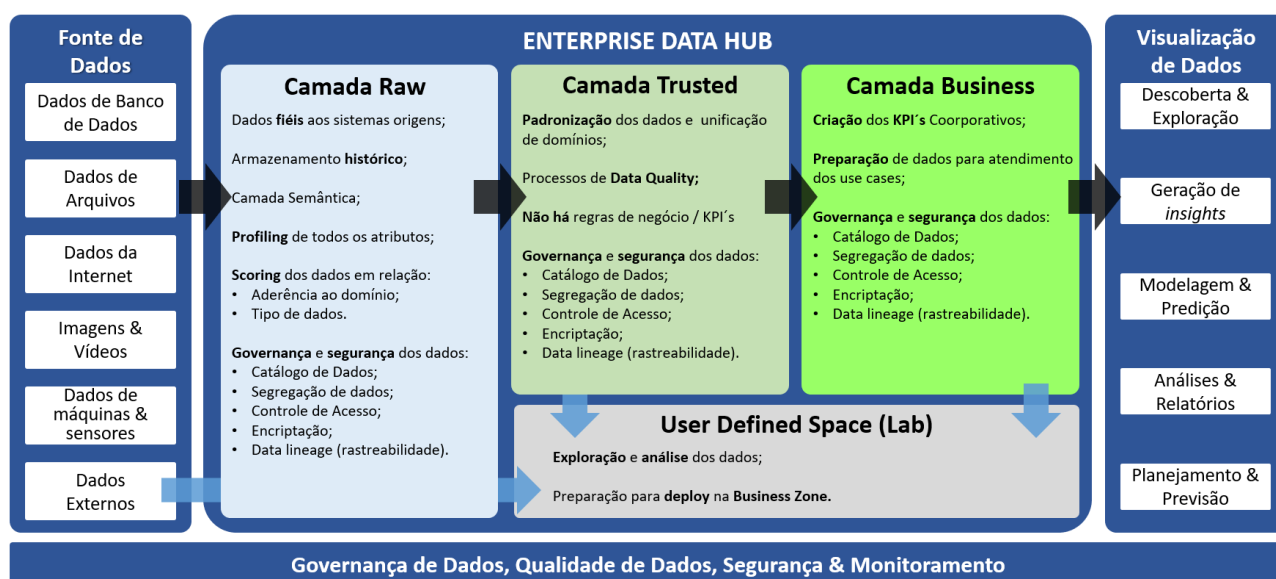


Figura 3 - Arquitetura geral de dados do Framework CNI

Os objetos descritos na **Tabela 1** foram definidos de acordo com a Arquitetura Geral de Dados do framework CNI apresentado na **Figura 3**.

Tabela 1 - Arquitetura geral de dados do framework CNI

Camada/ Diretório	Sigla	Schemas (com um ou mais níveis)	Tabelas	Exemplo
Raw	raw	<ul style="list-style-type: none"> ▪ Tipo de origem: Crawlers (crw), Banco de Dados OLTP tipo de origem (bdo) ou Arquivo de usuário (usr) e; ▪ Nome da fonte: Exemplo: nome da fonte crw/ibge ou bdo/protheus11 	<nome_original>	<ul style="list-style-type: none"> ▪ raw/crw/mte/rais_vinculo ▪ raw/bdo/protheus/akd010 ▪ raw /usr/oba/prm_cod_detalhamento_ne g ocio_x_cr
Trusted	trs	Natureza das entidades de negócio: Master Data (mtd) natureza das entidades de negócio para tabelas de cadastro e referência, ou Evento (evt) para tabelas de eventos de negócio. E se existirem muitas tabelas, pode ser interessante criar alguns schemas específicos para as subject areas em questão. Exemplo: mtd/localidade ou evt/producao	<entidade_negocio>_<f / p / v / a > f = substituição full (truncate /insert) p = substituição parcial (delete /insert) v = versionamento (merge /exchange partition) a = append (insert)	<ul style="list-style-type: none"> ▪ trs/mtd/localidade/uf_f trs /mtd/localidade/municipio_f ▪ trs/mtd/cnae/suclasse_f trs /evt/producao/atendimento_a
Business	biz	Visão/assunto de negócio	<ul style="list-style-type: none"> ▪ dim /dmh_<assunto>_<detalhe>, onde dim = dimensão ou dmh = dimensão histórica ▪ fte /fta_<assunto>_<detalhe>_<fil tr o>, onde fte = fato elementar (ex. qtd = 1) ou fta = agregada 	<ul style="list-style-type: none"> ▪ biz/loc/dim_localidade_uf, biz /loc /dim_localidade_municipio, biz/fin /fta_operacao_result_financ
User Defined Space	uds	Visão/assunto de negócio	A critério do usuário sugestão: <entidade>_<filtro>	-
Histórico Congelado para auditoria	hst	O mesmo schema da raw correspondente à tabela "congelada"	<nome_original>	<ul style="list-style-type: none"> ▪ hst/bdo/protheus/akd010

Veja na **Tabela 2** outros objetos auxiliares/complementares à Arquitetura de dados.

Tabela 2 - Objetos complementares à arquitetura de dados

Camada/ Diretório	Sigla	Schemas (com um ou mais níveis)	Tabelas	Exemplo
Landing Zone (área de transição)	lnd	<ul style="list-style-type: none"> ▪ Tipo de origem: Crawlers (crw), Banco de tipo de origem Dados OLTP (bdo) e, ▪ Nome da fonte: exceto de nome da fonte usuário que tem um container próprio por questões de permissão de acesso. 	<nome_original>	<ul style="list-style-type: none"> ▪ lnd/crw/mte/rais_vinculo ▪ lnd/bdo/protheus/akd010 ▪ lnd/bdo /protheus/ct1010
User Landing Zone (área de transição)	uld	Se necessário, criar por área usuária responsável pela interface	<nome_original>	<ul style="list-style-type: none"> ▪ uld/uniepro /apuracao_201907
Archive (arquivos que passaram pelas landing zones e que não foram rejeitados)	ach	Usar o mesmo schema da landing aonde chegou o arquivo, pode ser a landing estruturada (lnd) ou a landing de usuario (uld). E adicionar um subdiretório com o timestamp (somente números) do processo (o mesmo do dh_insercao_raw)	Mesmo nome usado na landing zone em que chegou.	<ul style="list-style-type: none"> ▪ ach/lnd/crw/mte /rais_vinculo /20191223102345 ▪ ach/lnd/bdo/protheus /akd010 /20191005101332 ▪ ach/uld/uniepro /apuracao_201907 /20190522120000
Arquivos Rejeitados que passaram pelas landing zones	rjt	usar o mesmo schema da landing aonde chegou o arquivo	Mesmo nome usado na landing zone em que chegou, sufixado por _rjt	<ul style="list-style-type: none"> ▪ rjt/crw/mec /enad_2019_rjt
Registros rejeitados no Data Quality	err	O mesmo schema da raw correspondente para os registros corretos	O mesmo da raw correspondente para os registros corretos, sufixado por _err	<ul style="list-style-type: none"> ▪ err/crw/mec /enad_2019_err

Log	log	Não é necessário abrir por schema	<nome do projeto>_<nome do processo>	-
Controle de carga	mgt	Próprio. Não se aplica (NA)	Próprias. NA	▪ NA
Catálogo Governança (usando estrutura de Controle de Carga)	-	<ul style="list-style-type: none"> ▪ /mgt/raw/gov ▪ /mgt/trs/governanca ▪ /mgt/biz/governanca 	Próprias. NA	▪ NA

Outros objetos auxiliares/complementares à Arquitetura de dados, com nomenclatura sugerida acima, e outras como temporária (tmp). O importante é que sejam definidos a priori a nomenclatura dos objetos que farão parte do projeto. (Tabela 2)

2.3 Campos

As tabelas correspondentes às origens ingeridas no data lake (raw zone) deverão manter os nomes de campos e tipos originais, os padrões a seguir valem para as tabelas criadas na estrutura do data lake para atender o negócio, nas camadas trusted, business e user defined.

Usar natureza + substantivo [+ qualificador], exemplo: cd_municipio, vl_receita_orcada, nr_km_percorrido, sg_uf, cd_cnpj, cd_cep, cd_cep_formatado.

OBS: evitar usar acentuação, cedilha, espaços e outros caracteres especiais, recomendado apenas letras, números e underline/underscore.

2.3.1 Natureza

Tabela 3 - Natureza das Informações

ID	Natureza da informação	Prefixos	Observações
1	código	cd	<ul style="list-style-type: none"> ▪ Código definido pelo negócio para identificar um objeto. Para campo de tipo, usar cd_tipo. ▪ Não confundir com identificador. Existem códigos, por exemplo em tabelas de referência, que são identificadores, mas são considerados cd para efeito de nomenclatura, ex: cd_municipio. ▪ Quando for uma sigla existe uma natureza específica sg, usá-la no lugar de código, ex: sg_uf.
2	data	dt	Data sem hora. Em caso de separação em dia, mês, ano, ano/mês usar cd, ex: cd_ano, cd_ano_mes, cd_dia_semana, etc.
3	data/hora	dh	Timestamp, data com hora
4	nome	nm	Nome do objeto (não confundir com descrição)
5	descrição	ds	Usado quando o campo descreve ou conceitua um objeto (não confundir com nome)
6	flag	fl	Flag representando 1 para sim e 0 para não
7	hora	hr	Formato HH ou HH:MM, se for em decimal usar nr_hr...
8	identificador	id	Identificador, normalmente sistêmico (não confundir com código)
9	quantidade /unidades	qt	Inteiro, se composto com unidades específicas: qt_hr, qt_min, qt_seg, qt_km, etc
10	número /unidades	nr	Com decimais, se composto com unidades específicas: nr_hr, nr_km, nr_kg
11	sigla	sg	Usado quando o conteúdo é uma sigla. Ex: sg_uf
12	valor	vl	Para valores com casas decimais ou não
13	percentual	Pc	Decimal que representa percentual
14	taxa/fator	Tx	Taxa que representa a relação entre dois valores

15	surrogate key	Sk	Chave específica de soluções DW para dimensão, sequencial inteiro. Evitar o uso, criar somente se for indispensável.
16	sequencial	Sq	Sequencial inteiro usado quando há versionamento de ocorrências únicas de uma tabela, no tempo.
17	Tempo	tm	Para representação de tempo em HHHHH:MM:SS.sss para duração (se duração em decimal, usar nr_hr)
18	Key-value	kv	Campo cujo conteúdo é uma estrutura de chave e descrição (ex: json {{key:value},{key:value}, ...}) com n ocorrências.

2.3.2 Substantivos, Qualificadores e Abreviaturas

Quando da necessidade de abreviar algum substantivo ou qualificador que representa uma informação, deve ser documentado abaixo e consultado para que não sejam utilizadas diferentes formas para um mesmo conceito, a não ser que seja estritamente necessário.

Substantivos

Tabela 4 - Abreviatura de substantivos

ID	Substantivos	Abreviatura
1	Localidade	locl
2	Estabelecimento	estbl
3	Matrícula	matr
4	Operação	oprc
5	Centro de Custo	ccust
6	Produto	prdt
7	Receita	rcta
8	Despesa	dpsa
9	Departamento	depto
10	Origem	or
11	Delimitador	delim
12	Atualização	atualiz
13	Classificação	classif
14	Frequência	freq
15	Localização	localiz

Tabela 5 - Abreviatura de qualificadores

ID	Substantivos	Abreviatura
1	Orçado	orcd
2	Realizado	real
3	Original	orgl
4	Operacional	opcl
5	Recebido(a)	recb
6	Formatado(a)	fmttd
7	Responsável	resp

Tipo

Tabela 6 - Tipos de campo

ID	Tipo de campo	Tamanho	Padrão para banco
1	Texto	até 5	char(n)
2	Texto		varchar(n)
3	Número Inteiro		int
4	Número Inteiro		bigint

5	Número Inteiro		numeric(n,0)
6	Número Decimal		numeric(n,d)
7	Flag (1, 0 ou NULL)		bit ou int
8	Data (YYYY-MM-DD)		date
9	Data e Hora (YYYY-MM-DD HH:MI:SS.sss)		datetime
10	Hora (HH ou HH:MM)		char(5)

2.3.4 Campos de controle

Todas as tabelas RAW deverão conter, além dos campos recebidos das respectivas fontes, os seguintes campos de controle:

- nm_arq_in: nome do arquivo de entrada com extensão, quando a origem não for sistêmica, senão NULL
- nr_reg: número da linha em que o registro se encontra no arquivo de entrada, quando a origem não for sistêmica
- dh_arq_in: timestamp do arquivo de entrada (propriedades), quando a origem não for sistêmica
- dh_insercao_raw: data de início do processo de carga válido para o fluxo completo de execução que for definido, gravado em todos os registros inseridos no data lake.
- kv_process_control: informações de execução da carga do registro no data lake.

Todas as tabelas TRUSTED e BUSINESS devem conter os seguintes campos de controle:

- dh_insercao_trs: data de início do processo de carga válido para o fluxo completo de execução que for definido, gravado em todos os registros inseridos no data lake.
- dh_insercao_biz: data de início do processo de carga válido para o fluxo completo de execução que for definido, gravado em todos os registros inseridos no data lake.
- kv_process_control: informações de execução da carga do registro no data lake.

2.4 Valores padrão para métricas órfãs

No nível analítico podem existir métricas cuja dimensão referenciada não tem o valor referenciado em duas situações: não se aplica à métrica ou não informada na origem. Para esses casos, as informações originalmente nulas devem ser preenchidas com os seguintes conteúdos a depender do tipo de campo:

- Não informada: -98 (int), '98' (char de 2), 'N/I' (char de 3 a 12), 'NÃO INFORMADA' (char >=13)
- Não se aplica: -99 (int), 99 (char de 2), 'N/A' (char de 3 a 12), 'NÃO SE APLICA' (char >=13)

2.5 Processos de carga no Data Lake Storage

- Fonte/Origem para RAW: (<fonte>_<nome_original>) org_raw_<nome da fonte_tabela raw>

Exemplo: org_raw_mte_rais_vinculo

- RAW para TRUSTED: (<assunto>_<detalhe>_< f / p / v / a >) raw_trs_<nome da tabela trusted>

Exemplo: raw_trs_cnae_suclasse_f

- TRUSTED para BUSINESS: (dim/dmh/fte/fta_<assunto>_<detalhe>) trs_biz_<nome da tabela refined>

Exemplo: trs_biz_dim_localidade_municipio, trs_biz_fta_gestao_financeira

- RAW para BUSINESS: (dim/dmh/fte/fta_<assunto>_<detalhe>) raw_biz_<nome da tabela refined>

Exemplo: raw_biz_dim_localidade_municipio, raw_biz_fta_gestao_financeira - Este caso é uma exceção onde fazemos a ingestão de dados não sistêmicos diretamente para a biz por não haver nenhum tratamento do dado.

- BUSINESS para BUSINESS: (dim/dmh/fte/fta_<assunto>_<detalhe>) biz_biz_<nome da tabela refined>

Exemplo: biz_biz_fta_gestao_financeira_kpi_pivot

- WORKFLOW responsável pelo encadeamento dos processos de carga formando o ciclo de ingestão e tratamento da informação da sua origem até o seu destino: wkf_<nome do assunto de negócio>

Exemplo: wkf_atendimento, wkf_controle_financeiro, wkf_vendas

3. Sistemas internos- Padrões de desenvolvimento – Ingestão de dados na camada RAW

O processo de ingestão de dados pode ser estruturado a partir de fontes OLTP (sistemas internos)

3.1 Fontes OLTP

Para fontes de sistemas transacionais, o processo de ingestão é estruturado por um framework desenvolvido para as necessidades da CNI.

Damos o nome framework para a implementação que na verdade é composta por templates parametrizados de execuções de pipelines no ADF, garantindo homogeneidade e segurança no desenvolvimento. O resultado, de maneira geral é que para a adição de uma nova fonte é necessária apenas a configuração via parâmetros de uma instância do template de cargas.

3.1.1 Pré-requisitos para os Desenvolvedores

- JSON
- SQL

3.1.2 Pré-requisitos CNI-STI

Antes de criar a demanda aos desenvolvedores e conceder o acesso ao ADF, o time de sustentação STI deve se atentar a três importantes pontos no framework de carga:

1) É um RDBMS que já foi utilizado?

Até o momento de escrita deste documento, como tecnologias RDBMS para fontes de dados, há conexões para Oracle (11g e 12c), MS SQLServer (2017+, Azure SQLDB, Synapse), MySQL e PostGres. Caso a nova origem esteja suportada por uma outra tecnologia, como DB2, por exemplo, é necessário criar um Linked Service no ADF.

2) A conexão ao RDBMS foi incluída no template de cargas raw?

Os parâmetros de conexão de RDBMS são declarados em um pipeline específico do ADF, em `templates/raw/bdo/raw_load_bdo_unified/raw_load_bdo_unified__0__switch_env`. O parâmetro dos ambientes dev e prod deve ser configurado.

3) Nos templates de carga, há a implementação dos tipos de carga requeridos considerando este RDBMS?

No framework, após a etapa `raw_load_bdo_unified__2__switch_db`, o processo segue por segmentações que consideram as particularidades de cada RDBMS para a implementação das queries de carga.

Em `templates/raw/bdo`, deve haver um diretório com o nome do RDBMS da origem e, dentro dele, as implementações dos tipos de carga atualmente disponíveis para essa tecnologia. Caso não esteja nesse diretório, é necessário criar a implementação. A criação dessa implementação, caso se faça necessário, deve ser direcionada à equipe de sustentação da STI-CNI.

3.1.3 Desenvolvimento

Primeiramente, crie uma nova *branch* de trabalho no Azure Data Factory. Para exemplificação deste procedimento, utilizarei a *branch doc__data_ingestion*.

Os processos de carga de origem de fontes transacionais são mantidos no diretório `/raw/dbo/<nome do sistema>`. Caso o nome do sistema não esteja previamente criado na estrutura de diretórios, crie uma pasta com o nome do sistema, todo em letras minúsculas. Para este exemplo, utilizaremos um sistema denominado *systemx*.

Abra o diretório de qualquer sistema previamente disponível e copie qualquer um dos processos já implementados – o título do processo deve conter o prefixo `org_raw_`. O nome do processo no diretório do sistema de origem é composto por:

`org_raw_<nome do sistema>_<nome da tabela>`

Assumindo a tabela `table1` para o nosso exemplo, o pipeline será nomeado `org_raw_systemx_table1`. Após estabelecer o nome correto para o processo, verifique que a implementação consiste na execução de um subpipeline denominado `raw_load_bdo_unified__0__switch_env` e a definição de dois parâmetros: *tables* e *env*.

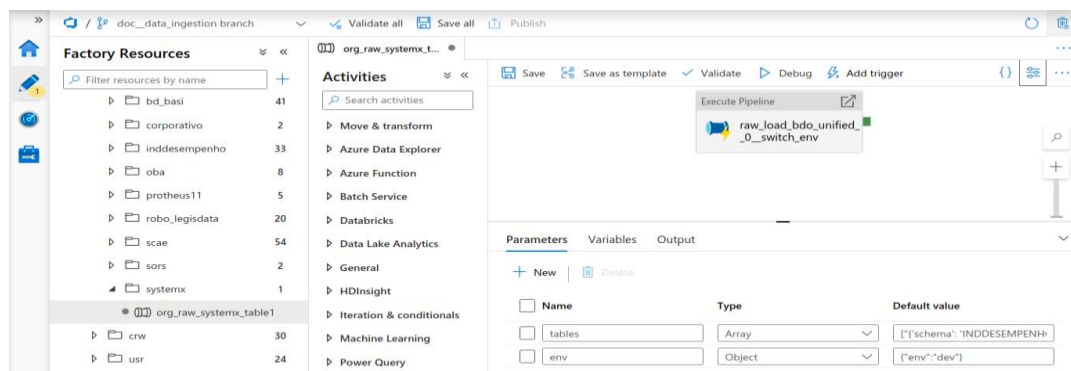


Figura 18

Na atividade de execução de pipeline definida por *raw_load_bdo_unified_0_switch_env*, nada deve ser modificado. Nosso trabalho consistirá basicamente em alterar o parâmetro *tables*, pois é este parâmetro que define todo o processo de carga. O parâmetro guarda um objeto do tipo *array*, mas este *array* possui apenas um item, um objeto, que guarda dentro de sua estrutura chave-valor todas as diretrizes necessárias para a implementação da carga até a camada *raw* no data lake.

O parâmetro *tables* pode ser definido conforme o exemplo a seguir:

```
[{"schema": 'INTEGRADORWEB',
'table': 'TB_CONGELADO_STI', 'is_history': 'True',
'load_type': 'incremental',
'control_column': 'DS_ANO_MES',
'control_column_type_2_db': 'datetime',
'control_column_default_value': '20160101000000',
'control_column_mask_value': 'MMYYYY',
'partition_column': 'DS_ANO_MES',
'partitions': 12,
'columns':
'CD_PRODUTO_SERVICO,DS_PRODUTO,CD_CATEGORIA,DS_CATEGORIA,CD_LINHA,DS_LINHA,DT_I
NICIO,DT_TERMINO_PREVISTO,DT_TERMINO,FL_ORIGEM_RECURSO_MERCADO,FL_ORIGEM_REC
URSO_FOMENTO,FL_ORIGEM_RECURSO_SENAI,FL_ORIGEM_RECURSO_OUTRASENT,FL_ATENDIMEN
TO_REDE,ORIGEM_RECURSO,ATENDIMENTO_EM_REDE,DRS_ATENDIMENTO_REDE,MUNICIPIO_AT
ENDIMENTO,VL_FINANCEIRO,VL_ECONOMICO,VL_PRODUCAO,QT_HORAS_PREVISTAS,QT_ENSAIOS
```

```

_PREVISTOS,QT_CALIBRACOES_PREVISTAS,QT_MAT_REF_PREVISTOS,QT_RELATORIOS_PREVISTOS,
QT_HORAS_REALIZADAS,QT_ENSAIOS_REALIZADOS,QT_CALIBRACOES_REALIZADAS',
'raw': {
  'partition_by': [
    {'col_name': 'YEAR',
'implementation': 'df.withColumn(\\'YEAR\\',
functions.substring(functions.col(\\'DS_ANO_MES\\'), 3, 6)).withColumn(\\'YEAR\\',
functions.col(\\'YEAR\\').cast(\\'int\\'))'
    },
    {'col_name': 'MONTH', 'implementation':
'df.withColumn(\\'MONTH\\', functions.substring(functions.col(\\'DS_ANO_MES\\'), 1,
2)).withColumn(\\'MONTH\\', functions.col(\\'MONTH\\').cast(\\'int\\'))'
    }
  ]
}
}"]

```

Para este exemplo temos uma `control_column = 'DS_MES_ANO'`, que é uma coluna do tipo `INTEGER`. Como o sistema de origem dessa tabela é Oracle, utilizamos `control_column_mask_value = 'YYYYMM'` e dessa forma, `control_column_type_2_db` assume `'datetime'`, o que permitirá tratar esse valor como data/hora, assumindo a máscara declarada. É importante lembrar que essa funcionalidade só está implementada para fontes Oracle e MySQL.

Vamos entender cada uma das opções de configuração.

schema

Tipo: string

Obrigatório: sim

Descrição: nome do schema no banco de dados de origem que contém a tabela de dados de interesse.

Valores aceitos: nome do schema, em maiúsculo ou minúsculo.

table

Tipo: string

Obrigatório: sim

Descrição: nome da tabela no sistema de origem.

Valores aceitos: nome da tabela em maiúsculo ou minúsculo.

load_type

Tipo: string

Obrigatório: sim

Descrição: tipo de carga

Valores aceitos: 'incremental', 'full'. Para Oracle há também 'full_balance', 'year_overwrite', 'incremental_with_join'.

control_column

Tipo: string

Obrigatório: sim, para os casos em que *load_type* assume os valores: 'incremental', 'incremental_with_join'

Descrição: nome da coluna de controle de incrementos (delta). Através dela, verificamos a existência de novos dados na tabela de origem.

Valores aceitos: nome da coluna em maiúsculo ou minúsculo

control_column_type_2_db

Tipo: string

Obrigatório: sempre quando *control_column* é declarado

Descrição: informa o data type utilizado pela coluna de incremento. No momento, a implementação permite 2 valores para essa chave:

- datetime: utilizado para todos os tipos data – timestamp, date, datetime, etc – independente da tecnologia do banco de dados de origem. No caso da coluna de controle se do tipo data ou data/hora, datetime deve ser utilizado

- bigint: utilizado para os tipos numéricos – integer, bigint, numeric, decimal, etc – independente da tecnologia do banco de dados de origem. Para os tipos numéricos, com exceção de ponto flutuante, declare bigint.

Para origens Oracle ou MySQL, é possível utilizar o valor datetime para colunas de controle de qualquer tipo na origem, mas que tenham representação de uma data ou data hora. Por exemplo, numa origem MySQL, a coluna de controle de carga para um determinada tabela é do tipo INTEGER e guarda valores no formato 20210105, uma representação numérica para YYYYMMDD. Dessa forma, pode-se declarar `control_column_type_2_db = 'datetime'` e em seguida, `control_column_mask_value = '%Y%m%d%H%i%s'` (a chave `control_column_mask_value` é descrita nas seções a seguir). O valor da máscara para interpretação do dado como uma data/hora é específico para cada tecnologia de banco de dados; por isso, esteja atento à formatação de data de cada tipo de origem.

Valores aceitos: 'datetime', 'int', 'string'

Valor padrão sugerido: 'datetime'

control_column_default_value

Tipo: string

Obrigatório: sempre quando *control_column* é declarado

Descrição: String que representa o valor padrão a ser utilizado para `control_column`, assumindo o tipo declarado em `control_column_type_2_db`. Esse valor serve como limite inferior para as cargas históricas e também como valor padrão no `NVL()` dos incrementos. Quando `control_column_type_2_db = 'datetime'`, o valor dessa coluna assume notação segundo a máscara 'YYYYMMDDHHmmSS', pois todos os watermark de data ou data/hora são persistidos com essa notação na tabela de controle de cargas. Exemplo: '20210101143000'

Valores aceitos: strings compatíveis com o tipo declarado em `control_column_type_2_db`; Exemplo: o tipo declarado é bigint. O valor em `control_column_default_value` é '0'

control_column_mask_value

Tipo: string

Obrigatório: sempre quando *control_column* é declarado

Descrição: máscara a aplicar sobre os dados de *control_column* na origem para permitir a conversão para o tipo declarado em *control_column_type_2_db*. No projeto, o uso mais amplo é o que possibilita a conversão de *control_column* para 'datetime'.

Valores aceitos: qualquer *string* que represente uma máscara de conversão aceita por CAST().

partition_column

Tipo: string

Obrigatório: não

Descrição: nome da coluna na origem que permite particionar os dados para a carga, isto é: a partir da segmentação lógica dos dados nessa coluna, estabelecer conexões múltiplas ao *RDBMS* de origem e garantir alto paralelismo na transferência de dados para o *Data Lake* na *Azure*. Essa coluna deve ser **obrigatoriamente** do tipo *INTEGER* e quanto mais próxima de uniforme for a distribuição, melhor o resultado da aplicação do particionamento.

Valores aceitos: nome da coluna em maiúsculo ou minúsculo

partitions

Tipo: integer

Obrigatório: sempre quando *partition_column* é declarado

Descrição: número de partições (segmentos lógicos) a aplicar sobre a coluna definida em *partition_column*. Esse número corresponde ao número de conexões paralelas utilizadas para o tráfego de dados.

Valores aceitos: qualquer número inteiro. É importante conhecer os limites da origem e avaliar cada caso para encontrar o melhor balanço *bytes/partição*.

columns

Tipo: string

Obrigatório: sim

Descrição: *string* com nomes das colunas na origem a serem trazidas na cláusula SELECT do procedimento de carga. O valor dessa chave deve ser declarado como uma única *string*, separando os nomes das colunas por vírgula. Evite o uso de espaços para economia de caracteres, pois há um limite de 8600 caracteres para o parâmetro *tables*. Exemplo de utilização: 'col1,col2,col3,col4'

Valores aceitos: *string* concatenando o nome das colunas conforme descrição acima. Os nomes podem estar em maiúsculo ou minúsculo.

raw

Tipo: dicionário

Obrigatório: não

Descrição: dicionário que contempla instruções adicionais para o processo de carga raw. Geralmente essas instruções são executadas pelo *Databricks*. O dicionário é apenas um *container* para um novo conjunto de instruções. A maioria delas reflete instruções específicas como particionamento ou *coalesce()* dos dados no *Databricks*.

Valores aceitos: os valores descritos aqui são as chaves atualmente disponíveis para uso no dicionário *raw*: 'partition_by', 'coalesce'.

raw.partition_by

Tipo: Array

Obrigatório: não

Descrição: Quando for necessário particionar os dados para escrita na camada *raw*, *raw.partition_by* é declarado conforme o exemplo a seguir:

```
[  
{'col_name': 'YEAR',
```



```
'implementation': 'df.withColumn(\'YEAR\', functions.substring(functions.col(\'DS_ANO_MES\'), 3, 6)).withColumn(\'YEAR\', functions.col(\'YEAR\').cast(\'int\'))'
}
```

Na base da estrutura temos um *array*, se seus itens são dicionários, para o qual as chaves são: *col_name* e *implementation*.

col_name é uma *string* do nome da coluna a ser utilizada para o particionamento. Se a coluna não existir nativamente na origem de dados, sua implementação pode ser declarada através da chave *implementation*, para a qual o valor é uma *string* contendo a instrução *pySpark* da sua implementação, que assume, **obrigatoriamente**, que essa operação será realizada em um *DataFrame* denominado *df*. Caso seja necessário fazer uso de funções do módulo *pyspark.sql.functions*, utilize o *namespace functions* para acessá-lo.

Aspas simples ou duplas dentro da instrução de implementação devem ser escapadas por `\\`.

Múltiplas colunas podem ser utilizadas para o particionamento. Para isso, declare em *raw.partition_by* quantos itens forem necessários. A hierarquia de particionamento para escrita na camada *raw* é aplicada na mesma ordem em que as definições são inseridas em *raw.partition_by*. Se temos a declaração na seguinte ordem (considerando *col_name*) *col1,col2,col3*, a hierarquia de partição do objeto *raw* será *col1,col2,col3*.

Valores aceitos: *array* de dicionários {'col_name': <valor>, 'implementation': <valor>}

raw.coalesce

Tipo: integer

Obrigatório: não

Descrição: número de partições – arquivos parquet - em que os dados serão escritos no Data Lake. Para tabelas sem a instrução *raw.partition_by*, o número de arquivos parquet escritos corresponde ao número declarado em *raw.coalesce*.

Para tabelas em que há a declaração de *raw.partition_by*, o valor declarado em *raw.coalesce* será o número de arquivos parquet adicionados ou sobrescritos em cada partição.

3.1.4 Testes de execução

Para testar a execução do seu pipeline, verifique a sintaxe do parâmetro *tables* e certifique-se que o parâmetro *env* = `{"env": "dev"}`. Em seguida, na barra superior, clique em *Debug*.

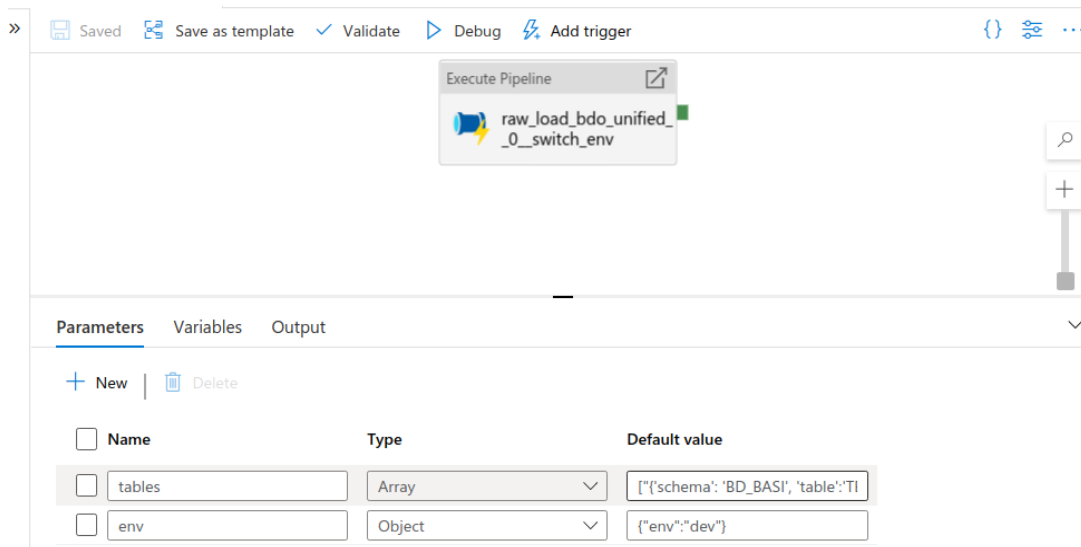


Figura 19

Na aba de confirmação, verifique novamente os parâmetros. Altere-os se necessário. Em seguida, clique em OK.

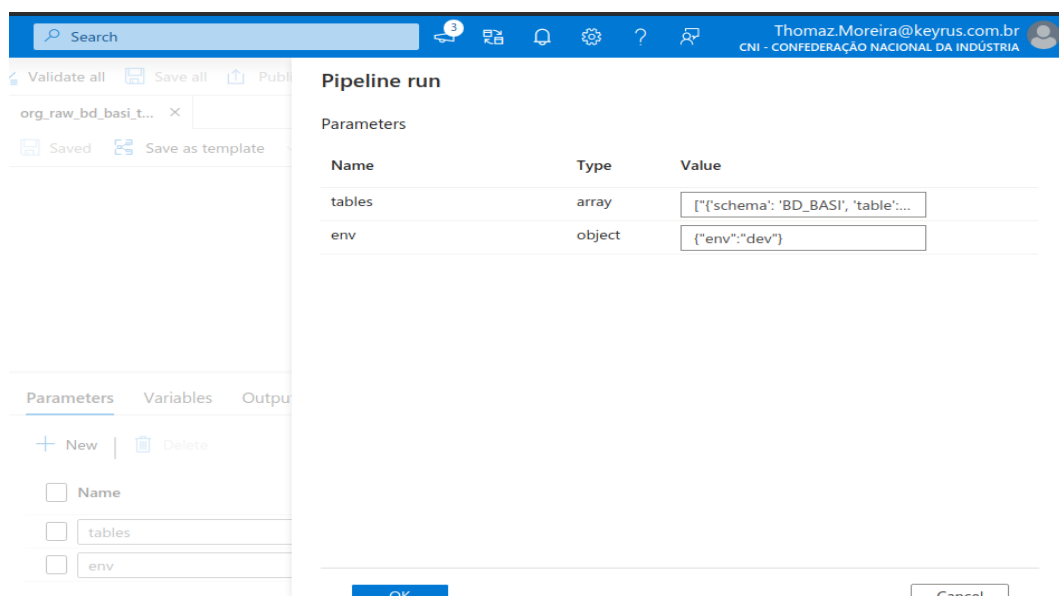


Figura 20

Clique em qualquer ponto da área branca do *canvas* de desenvolvimento, e monitore a sua execução através da aba *Output*, que surgirá ao lado de *Variables*.

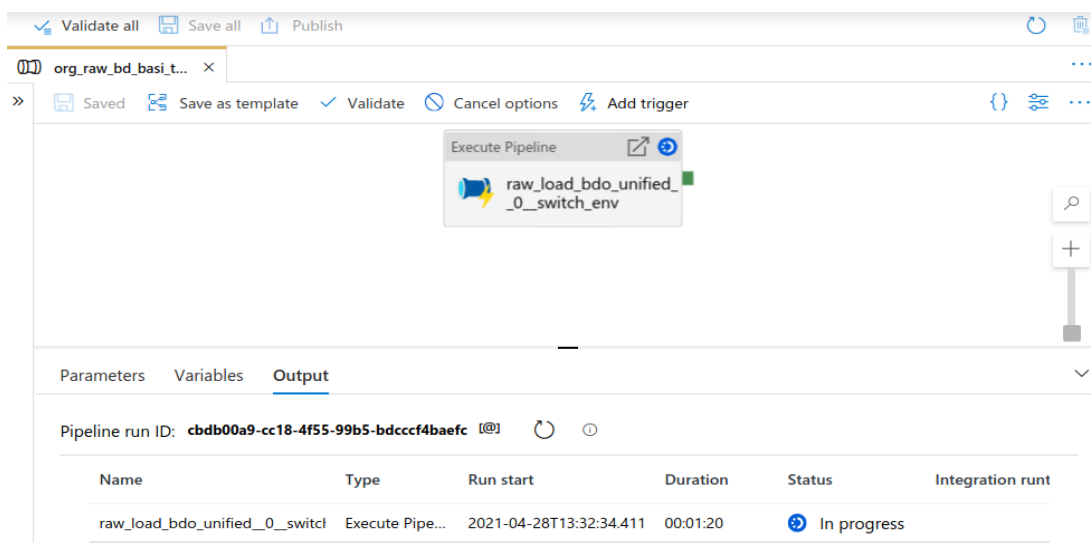


Figura 21

3.1.5 Publicação em produção

O procedimento de publicação em produção é o mesmo que está descrito para a seção *Crawlers*, considerando o repositório git *ENSI-BIGDATA*. A fim de evitar repetições no texto, consulte o procedimento na seção indicada.

3.1.6 Camada RAW

Para fontes OLTP, o procedimento anteriormente descrito compreende a cópia dos dados para a camada *landing (Ind)* e sua carga na camada RAW. O framework de cargas desenvolvido no *Azure Data Factory* garante a execução desse processo ponta a ponta através de um único ponto de implementação *Azure Databricks*. Assim, as tabelas de origem OLTP não necessitam de um notebook para a conformação na RAW, o pipeline ADF, corretamente parametrizado, é a única tarefa de desenvolvimento para esse caso.

4. Crawler / Bots (qualquer programa que busca dados externos)- Padrões de desenvolvimento – Ingestão de dados na camada RAW

Para fontes externas, o processo de aquisição e ingestão dos dados possui uma estrutura definida e padronizada através do uso de *bots* que são executadas em um *Azure Container Instance*. Porém, a implementação de cada *bot* depende exclusivamente das características únicas de cada fonte e da estratégia necessária para a aquisição dos dados. Ainda assim, os *bots* se adequam a uma estrutura de código bem definida, que é escopo deste documento.

4.1 Configuração do ambiente local no cofre de senhas

No portal do Azure Devops, o desenvolvedor deve gerar suas credenciais git e, em seguida, clonar em sua máquina o repositório *ENSI-BIGDATAACRAWLER* utilizando o *PyCharm* ou *VS code*. Isso pode ser feito através da url: https://CNI-STI@dev.azure.com/CNI-STI/ENSI-BIG%20DATA/_git/ENSI-BIGDATAACRAWLER.

Após clonar o projeto, o desenvolvedor receberá as credenciais para uso dos recursos compartilhados Azure. Nas credenciais, poderão ser identificadas as seguintes propriedades:

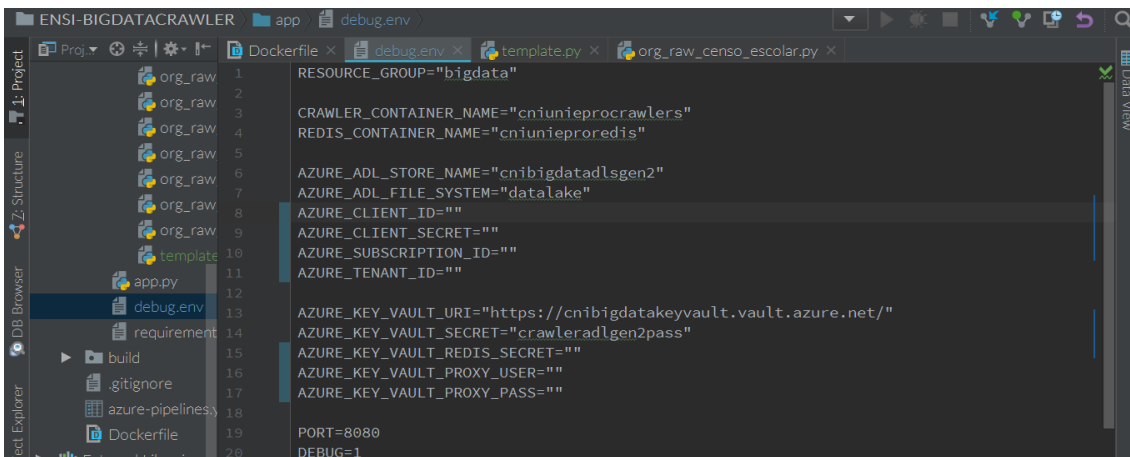
```
AZURE_CLIENT_ID
AZURE_CLIENT_SECRET
AZURE_SUBSCRIPTION_ID
AZURE_TENANT_ID
AZURE_KEY_VAULT_SECRET
AZURE_KEY_VAULT_REDIS_SECRET
AZURE_KEY_VAULT_PROXY_USER
AZURE_KEY_VAULT_PROXY_PASS
```

O desenvolvedor deve guardar essas informações de modo seguro, pois serão necessárias para a execução dos *bots* e dos testes e modo local.

O próximo passo é criar um *virtualenv* localmente para permitir o desenvolvimento. Escolha um diretório facilmente acessível para isso.

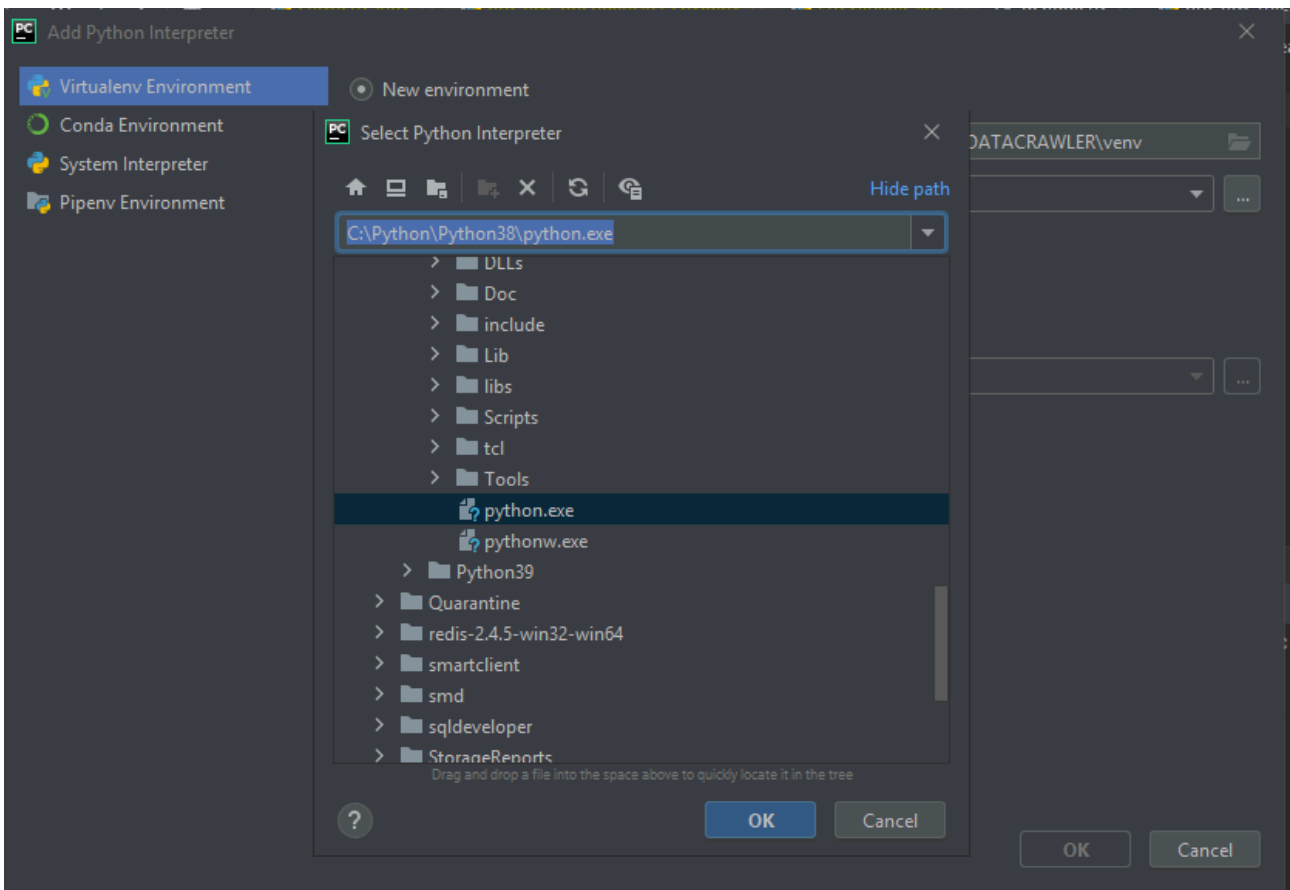
4.2 Desenvolvimento

Na IDE Python, abra sua *branch* de desenvolvimento.



```
1 RESOURCE_GROUP="bigdata"
2
3 CRAWLER_CONTAINER_NAME="cniunieprocrawlers"
4 REDIS_CONTAINER_NAME="cniunieproredis"
5
6 AZURE_ADL_STORE_NAME="cniunibigdataadlsgen2"
7 AZURE_ADL_FILE_SYSTEM="datalake"
8
9 AZURE_CLIENT_ID=""
10 AZURE_CLIENT_SECRET=""
11 AZURE_SUBSCRIPTION_ID=""
12 AZURE_TENANT_ID=""
13
14 AZURE_KEY_VAULT_URI="https://cniunibigdatakeyvault.vault.azure.net/"
15 AZURE_KEY_VAULT_SECRET="crawleradlsgen2pass"
16 AZURE_KEY_VAULT_REDIS_SECRET=""
17 AZURE_KEY_VAULT_PROXY_USER=""
18 AZURE_KEY_VAULT_PROXY_PASS=""
19
20 PORT=8080
21 DEBUG=1
```

Observação: no ambiente do cofre de senhas existem diversas versões do python instaladas no caminho “c:\Python”, na imagem estamos selecionando o Python 3.8 para a criação do venv.



No diretório *app/crawlers* há um arquivo denominado *template.py*. Esse arquivo contém o template básico de um *bot*, com todos os métodos obrigatórios da implementação, a listar:

- Condição inicial para desenvolvimento e testes locais:

```
if __name__ == '__main__':
```

- Função *execute()*, que expõe o *bot* à api:

```
def execute(**kwargs)
```

- Função *main()*, que provê a estrutura básica para a execução do crawler e todos os procedimentos necessários.

```
def main(**kwargs)
```

- Função *__call_redis()*, para criação, consulta e atualização dos pontos de retomada.

```
def __call_redis(host, password, function_name, *args)
```

- Função *authenticate_datalake()*, para autenticação no Data Lake, onde os dados são persistidos.

```
def authenticate_datalake() -> FileSystemClient
```

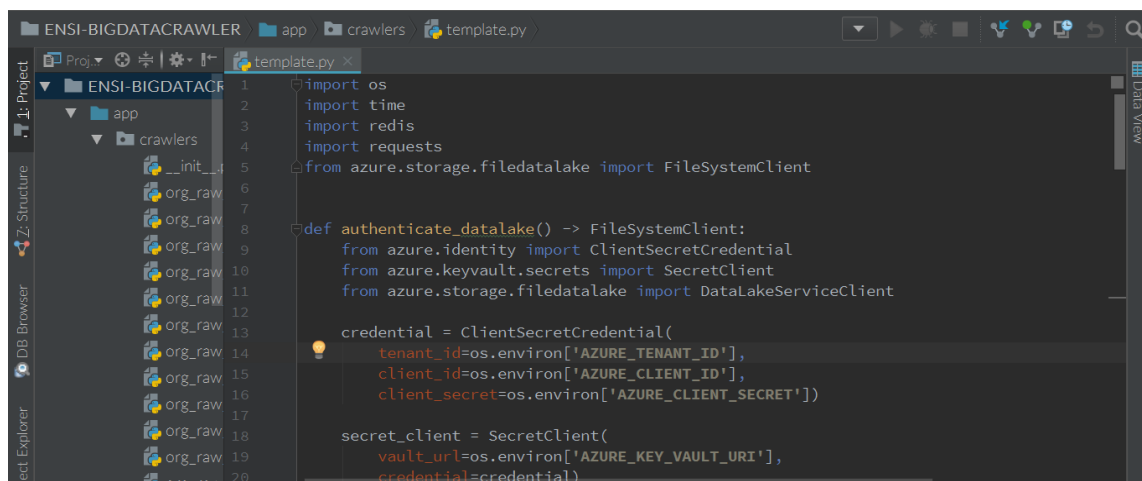


Figura 25

Mesmo durante o desenvolvimento, os dados devem ser persistidos no *Data Lake*. O código deve considerar todas as capacidades da entrega final, mesmo que a execução seja feita localmente. Na verdade, a aplicação AAD utilizada permitirá a escrita somente nos diretórios de desenvolvimento no *Data Lake*. A execução do *bot* na *Azure* deve prover exatamente o mesmo comportamento da execução local, exceto por segredos e parâmetros distintos.

Para iniciar o desenvolvimento do novo *bot*, copie o arquivo *template.py* e renomeie para o nome definido pela equipe STI na solicitação da demanda. Cada *bot* possui inúmeras especificidades em relação ao que é necessário para a aquisição dos dados. Caso seja necessário, consulte as outras implementações disponíveis para buscar ideias ou tirar dúvidas.

IMPORTANTE: os *bots* buscam os dados *as is* nas fontes e os persiste no *Data Lake* em um formato legível pelo *Apache Spark*. Nenhuma transformação deve ser aplicada aos dados, mas o formato deve ser adequado para atender a esse requisito. Exemplo:

- Arquivos compactados devem ser descompactados no tempo de execução do *bot*. Ficheiros não são diretamente legíveis pelo Spark.
- XLS e formatos binários não são nativamente legíveis pelo *Spark*, e, portanto, devem ser convertidos em *Parquet* ou *json* no tempo de execução do *bot* antes de serem persistidos no *Data Lake*. Nestes projetos, dizemos que sempre que possível, entregue os dados em *Parquet*, isso tende a facilitar todas as próximas etapas.

IMPORTANTE: Também é importante notar que com o uso do Redis para o controle do estado do crawler, não há a necessidade de manter o bot rodando por um longo tempo (mais do que 4 horas).

O desenvolvimento deve sempre garantir a retomada do estado de onde ele parou.

Dessa forma, mesmo que o pipeline seja cancelado por timeout, sempre será possível retomar a execução de forma consistente.

A única exceção a esse caso seria uma fonte de dados que muda em um intervalo mais rápido que 4h, nessa situação específica, teríamos que garantir a coleta completa dos dados em um intervalo menor que essa janela.

IMPORTANTE: Cada crawler deve retornar um código e mensagem do resultado da execução, sendo o código do resultado da execução 200 para *success*, 300 *is running* ou *without new files*, 500 para *error*. A seguir um exemplo de dicionário esperado pelo Azure data Factory como retorno de um crawler:

```
{
  "finished_with_errors": true,
  "exit": 500,
  "msg": "('Cannot connect to proxy.', timeout('The write operation timed out'))",
  "execution_time": 6579.410342931747,
  "ADFRerunRecoveryStatus": "None"
}
```

4.3 Testes de execução

Para executar/testar o *bot* localmente, basta executar o de implementação com o *virtualenv* ativo. Satisfeitos todos os procedimentos anteriores, o *bot* deve executar com sucesso e persistir os dados na área de desenvolvimento do *Data Lake*.

Utilizando a IDE, há inúmeras formas de configurar essa execução. Cada desenvolvedor deve averiguar essas possibilidades em sua IDE de preferência. De toda forma, é possível executar também via terminal. Tomarei o *bot template.py* como exemplo:

```
(virtualenv)$ python3.7 template.py
```

Se o *bot* opera de maneira adequada com as configurações de desenvolvimento/debug, então é possível prosseguir para a etapa de publicação em produção.

4.4 Resumo de crawlers

Nesse documento, entende-se que um *bot* é qualquer programa que busque dados externos por meio de *crawlers*, *scraping*, *api* etc.

Repositório = *ENSI-BIGDATACRAWLER*

Bom exemplo = *org_raw_ideb.py*

O bot é o responsável por capturar dados externos de algum site e depositá-lo no data lake, em um primeiro momento, na camada landing (*Ind*).

O que é feito nos bots:

- descompactação dos arquivos (ver função `__extract_files`)
- normalização dos dados (ver função `__normalize_str`)
- tratar da codificação do arquivo (ver função `__change_encoding`)
- conversão dos arquivos (*csv*, *tsv*, *txt* etc) no formato *parquet*
- eliminação de caracteres inválidos
- realizar o download do arquivo (ver função `__download_file`)

- utilizar o Redis, se for o caso, para guardar o estado da transferência dos dados (paginação) do site para a camada *Ind* (ver função `__call_redis`)

Após a etapa de gravação dos dados na *landing*, o fluxo deve chamar o script (notebook do Databricks) responsável pela gravação dos dados na *raw*.

Neste tipo de notebook (bom exemplo: *KEYRUS >> dev >> raw >> crw >> inep_ideb >> org_raw_ideb_ideb*) da *raw*, são utilizados os arquivos *PRM* (arquivo de parâmetros em Excel que serve também para documentar o bot).

Os arquivos *PRM* são utilizados no notebook do *Databricks* sendo responsáveis pela tradução (de-para) das colunas (nomes e tipos dos campos) do arquivo de origem para o arquivo que será gravado na *RAW*.

Observações para os arquivos *PRM* em nos nomes dos campos ("campo origem" e "campo destino"):

- não são permitidos espaços ==> trocar espaços em branco por underline
- não são permitidos caracteres especiais ==> trocar caracteres especiais por outros
- dar preferência a caracteres minúsculos ==> substituir maiúsculo por minúsculos

Esses cuidados visam manter a compatibilidade com o *HiveMetastore* e com o *Spark/Databricks*

Bom exemplo de arquivo prm:

`/prm/usr/inep_ideb/KC2332_IDEB_BRASIL_mapeamento_unificado_raw_V1.xlsx`

4.5 Publicação em DEV

A branch de DEV do repositório ENSI-BIGDATACRAWLER é responsável por receber os códigos implementados e por realizar o deploy dos bots nos containers de desenvolvimento.

Estes ambientes (containers) se assemelham ao de produção, assim tornando possível a realização de testes, consequentemente evitando erros no ambiente de produção.

O desenvolvedor deve criar sua branch baseada na master e realizar o desenvolvimento do bot, e, uma vez realizado o pull request para a branch DEV, poderá configurar seu pipeline no Data Factory

de LND e executá-lo com o parâmetro {"env": "dev"} e o nome do bot desenvolvido (o processo de desenvolvimento no ADF é bem descrito no próximo item deste documento).

New pull request

anttt_test into dev

Overview Files 1 Commits 1

Title

anttt_ti_pull_request

Description

Updated org_raw_anttt_ti.py

Markdown supported. Drag & drop, paste, or select files to insert. Link work items.

@ # **B** *I* `</>`

Updated org_raw_anttt_ti.py

Reviewers Add required reviewers

Search users and groups to add as reviewers

Work items to link

Search work items by ID or title

Tags

Create

Exemplo de pull request aberto da branch de operação para branch “dev”

Todos os usuários desenvolvedores possuem permissão para completar o pull request para “dev”, podendo ser utilizado como evidência de que o teste foi realizado.

Caso a execução apresente um erro de "bad request" não relacionado ao python implementado no bot, certifique-se que o ambiente de dev esteja ligado executando no Data Factory o pipeline "crawler_start" com o parâmetro {"env": "dev"}, e aguarde a conclusão (em torno de 5 minutos).

Para realizar a validação do pull request aberto para a master, a equipe da STI, sempre realizará uma execução em desenvolvimento antes de realizar a aprovação.

4.6 Publicação em produção

Para publicar em produção, através da IDE, adicione os arquivos necessários, faça o *commit* do código e em seguida a ação de *push* de sua *branch* local para a *branch* remota de mesmo nome. Esses são procedimentos padrões de versionamento com o *Git* e não carecem de detalhes neste documento.

Faça o login no *Azure DevOps* CNI-STI:

<https://dev.azure.com/CNI-STI/>

Clique no projeto ENSI-BIG DATA conforme apresentado na figura a seguir.

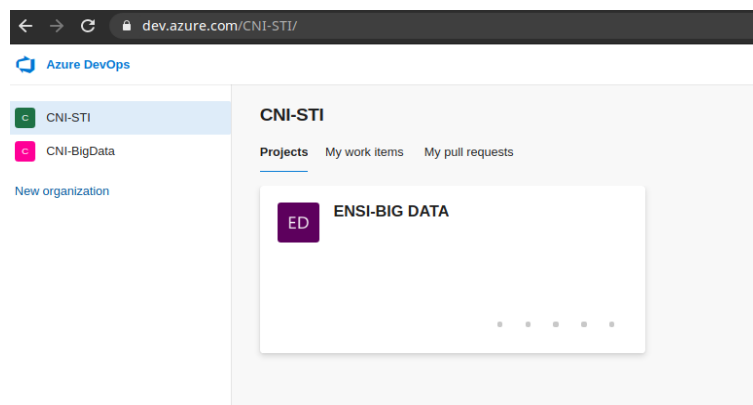


Figura 26

Na barra à esquerda, clique em *Repos*, depois *Pull Requests*. No seletor de repositórios da barra superior, escolha *ENSI-BIGDATA CRAWLER*. Em seguida, clique no botão *New Pull Request*.

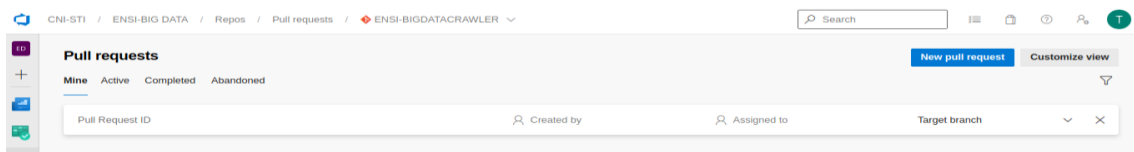


Figura 27

Selecione a sua *branch* para o lado esquerdo da operação e a *branch master* para o lado direito. Adicione um título, descrição e tags adequadas para a requisição. Antes de clicar em *Create*, verifique os arquivos e *commits* na barra superior.

CNI-STI / ENSI-BIG DATA / Repos / Pull requests / ENSI-BIGDATAACRAWLER

New pull request

feature__move_common_defs_to_modules into master

Overview Files 1 Commits 1

Title

Altered a comment to be able to document the merge procedure in DevOps.

Description

Altered a comment to be able to document the merge procedure in DevOps.

Markdown supported. Drag & drop, paste, or select files to insert. [Link work items.](#)

Altered a comment to be able to document the merge procedure in DevOps.

Reviewers Add required reviewers

Work items to link

Tags

crawlers Uniepro +

Create

Figura 28

Na tela a seguir, utilize a área de comentários para adicionar todas as evidências de trabalho. Inclua informações importantes e *screenshots* da execução. Se possível, adicione arquivos de *log* e outras evidências que permitem um nível fino de rastreabilidade. Caso haja itens de trabalho do *DevOps Boards* vinculados, adicione-os na área *Work Items* à direita. Referências a **cards do Trello** devem ser incluídas na descrição e nos comentários.

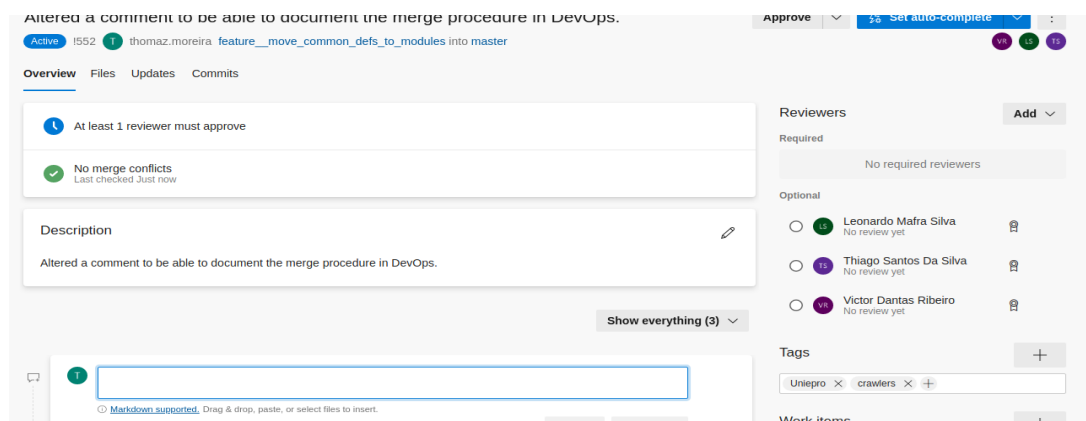


Figura 29

Após concluída essa etapa, é necessário aguardar pela aprovação ou pelas solicitações da equipe STI. Novas interações são alertadas via e-mail pelo próprio DevOps. Assim que concluída a solicitação, o *pull-request* é movido para a área *Completed*, na tela de acompanhamento.

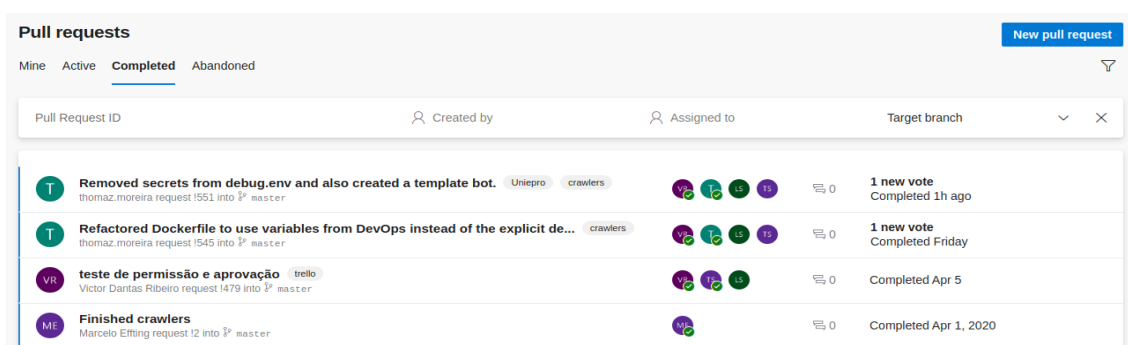


Figura 30

Em seguida, o pipeline de publicação das alterações em produção é automaticamente iniciado. E sua execução pode ser acompanhada através da opção *Pipelines*, na barra lateral esquerda.

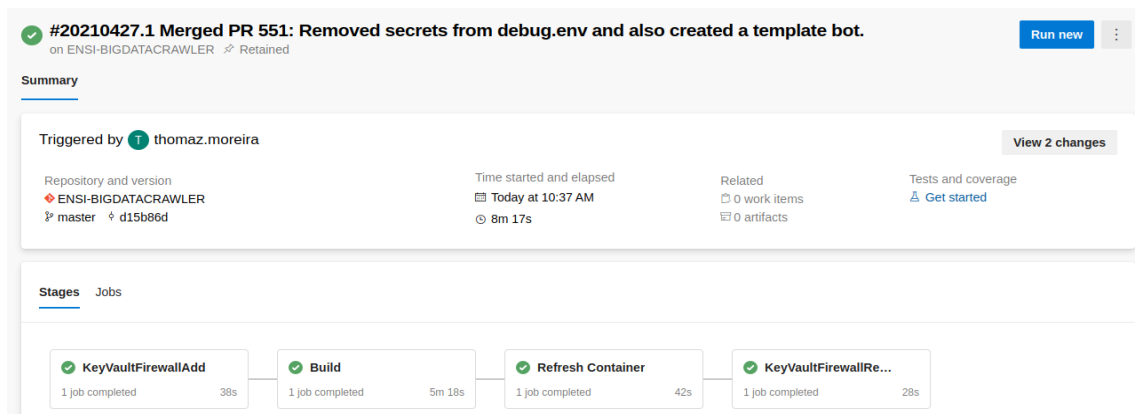
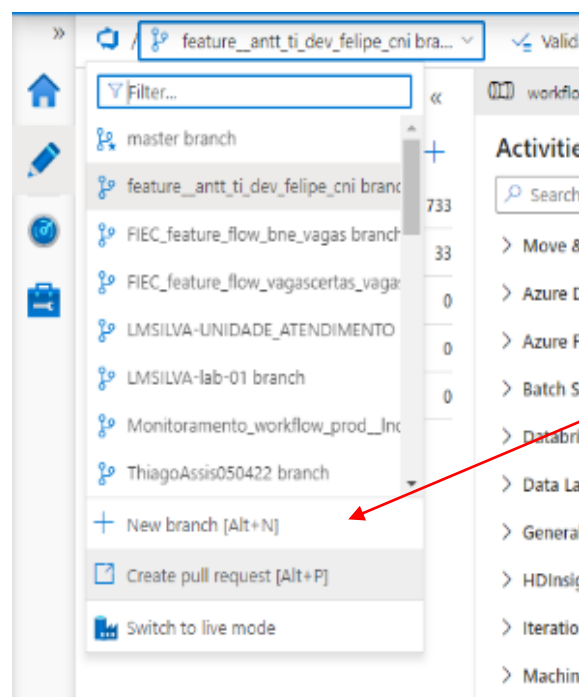


Figura 31

4.8 Desenvolvimento Azure Data Factory

No ADF, o desenvolvedor deve desenvolver seu pipeline responsável por realizar a execução do bot desenvolvido e publicado no repositório ENSI-BIGDTACRAWLER.

Primeiramente, o desenvolvedor deve criar uma nova branch:



O ambiente de desenvolvimento no ADF é organizado em diretórios, sendo que os principais deles são Ind/, raw/, trs/, biz/ e workflow.

Ind: Neste diretório, são desenvolvidos os pipelines de execução direta dos bots, ou seja, o pipeline responsável por executar o bot desenvolvido e publicado no repositório ENSI-BIGDATACRAWLER. Após executado, os dados serão escritos na camada Ind.

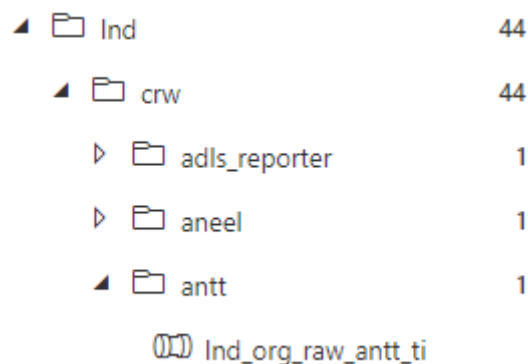
raw: Desenvolvimento dos pipelines responsáveis por executar os notebooks desenvolvidos via Data Bricks para realizar a ingestão dos dados para a camada raw

trs: Este diretório é responsável por persistir os pipelines de execução dos notebooks desenvolvidos via Data Bricks para a ingestão dos dados na camada Trusted.

biz: Pipelines de execução dos Notebooks de ingestão dos dados na camada Business.

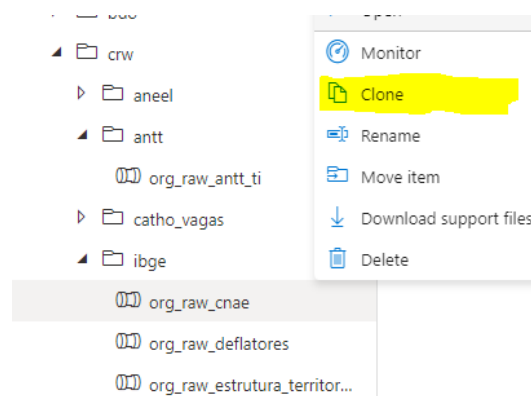
workflow: Nesta pasta, são desenvolvidos os workflows responsáveis por orquestrar a execução dos pipelines desenvolvidos.

Para começar a desenvolver nestes diretórios, basta criar uma subpasta com o nome do bot que está sendo desenvolvido dentro do diretório responsável pela atividade específica.



▲	Ind	44
▲	crw	44
▶	adls_reporter	1
▶	aneel	1
▲	antt	1
	Ind_org_raw_antt_ti	

A Estrutura por trás dos pipelines de sua respectiva pasta é a mesma, mudando apenas os parâmetros de execução que devem ser alterados pelo desenvolvedor de acordo com as informações de seu próprio bot ou notebook. Sendo assim, é possível realizar a clonagem de algum pipeline pertencente ao mesmo diretório, e alterar apenas seus parâmetros.



Após realizado o clone do pipeline, é necessário alterar os parâmetros de acordo com as informações do bot. No caso da camada LND, o pipeline necessita de dois parâmetros:

- **bot:**

Tipo: string

Descrição: string contendo o nome do bot a ser executado

Exemplo:

```
org_raw_antt_ti
```

- **env:**

Tipo: *object*

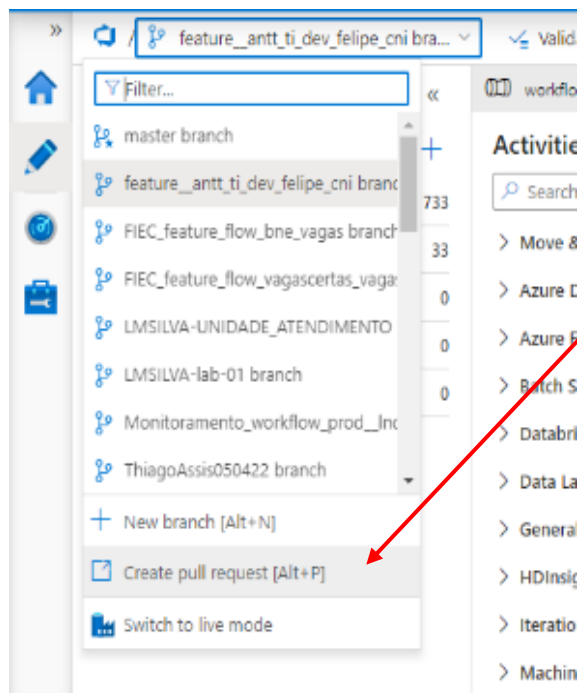
Descrição: dicionário com a chave *env* que informa o ambiente para execução do processo.

Exemplo:

```
{"env": "dev"}
```

Parameters Variables Settings Output			
<div><div>+ New</div><div> Delete</div></div>			
<input type="checkbox"/>	Name	Type	Default value
<input type="checkbox"/>	bot	String	org_raw_antt_ti
<input type="checkbox"/>	env	Object	{"env": "dev"}

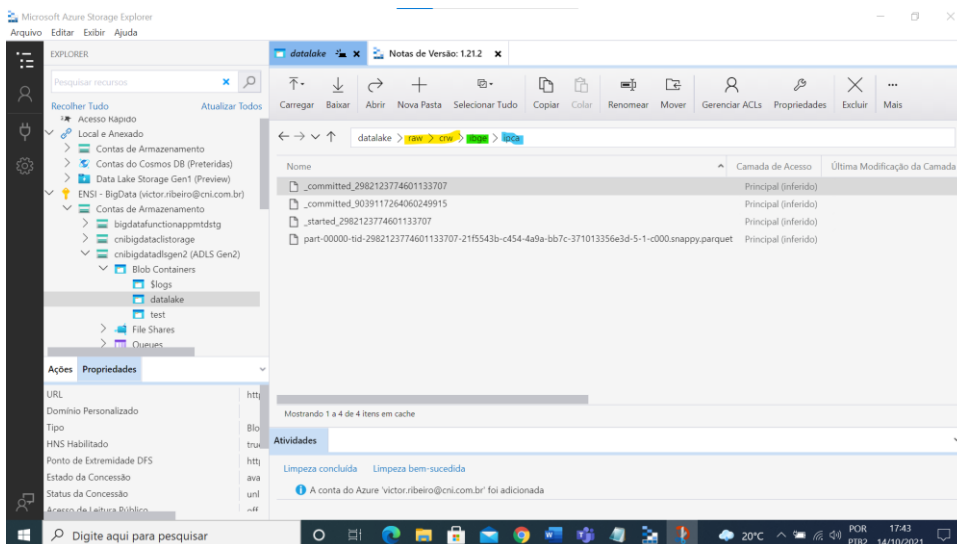
Por fim, basta validar o pipeline, salvá-lo, e acionar o botão debug, para realizar o teste e execução. Em seguida, deve-se fazer o pull request para publicação em produção.



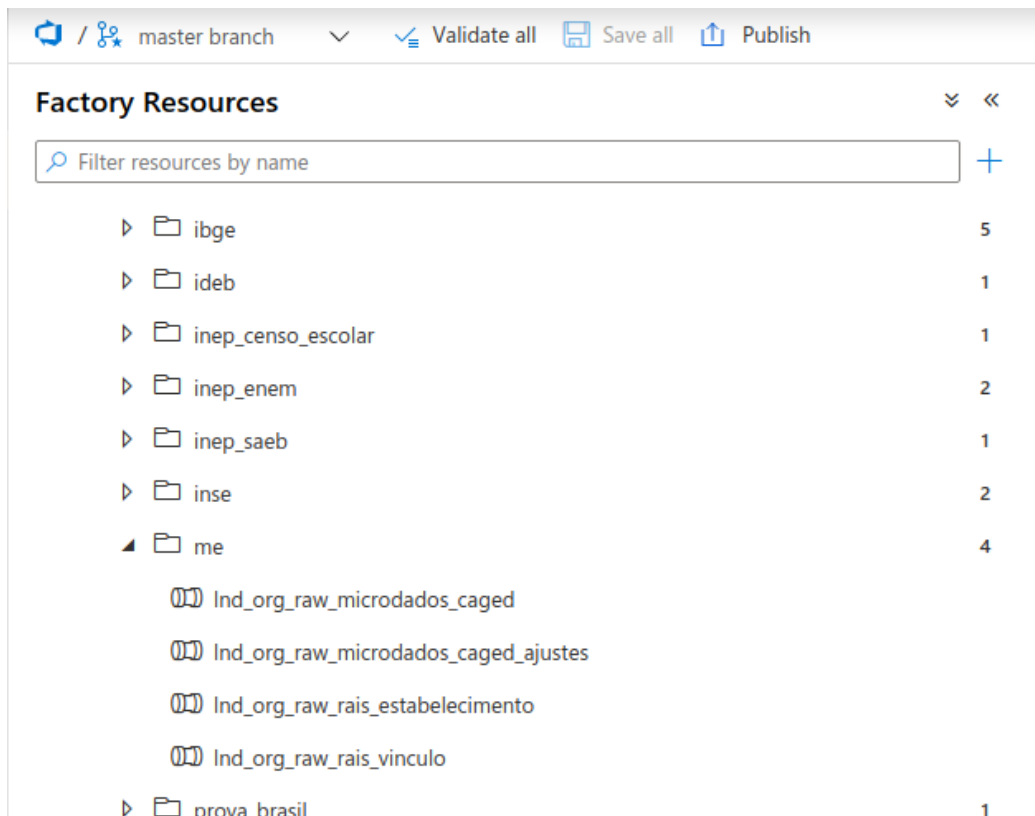
5. Camada RAW

O procedimento descrito até aqui traz os dados da fonte externa até a camada *landing (Ind)*. Como já compreendido, cada *bot* pode possuir uma implementação única e específica para buscar o conteúdo e lidar com as características singulares da origem. Como não há um padrão de como os dados são ofertados, segmentados e formatados, conformá-los em uma estrutura tabular na camada *RAW* exigirá uma análise específica de cada caso e pode envolver transformações muito complexas.

Sempre um processo/notebook de camada raw deve escrever os dados no **caminho** `/raw/crw/<Area de assunto do bot>__<table>` ou `/raw/crw/<Area de assunto do bot>/<table>`, pois o padrão de desenvolvimento vai buscar essa estrutura se os parâmetros forem preenchidos de forma correta. Veja os exemplos da imagem a seguir: Amarelo=caminho base, Verde=schema/Área de assunto do bot, Azul=table



Como uma boa prática, sugerimos que cada arquivo/tabela/schema na origem seja coletado por uma implementação distinta de bot. Como exemplo, temos para a base RAIS dois bots: *org_raw_rais_estabelecimento* e *org_raw_rais_vinculo*. Ambos buscam arquivos da mesma fonte/host, mas que estão em seções distintas do portal e possuem schemas distintos, pois representam entidades diferentes da RAIS. Os bots, embora muito semelhantes na sua implementação, realizam a coleta de dados específicos. No *Azure Data Factory*, cada bot é vinculado ao seu agendamento específico, que leva o nome do bot. Na execução dos agendamentos, é preciso identificar cada schema isoladamente e depurar as falhas individualmente.



Em um caso em que a coleta de vários objetos/schemas leve a retrabalho na implementação de *bots*, é possível unificar a coleta de todos os dados em um único *bot*, desde que na camada landing os objetos sejam entregues numa estrutura de diretórios que permita separar os schemas, considerando a recursividade de diretórios. Essa separabilidade é de extrema importância para o *Spark/Databricks* e o processo de adequação na camada *RAW*. Para este caso, obviamente no ADF só haverá uma implementação de processo *Ind* e não será possível depurar erros isoladamente.

Considerando estes pontos, o desenvolvedor deve buscar a estratégia que minimize o esforço e maximize a rastreabilidade. Essa decisão deve sempre ser aprovada pela equipe STI-CNI.

Os dados, após coletados e pousados na camada landing devem ser segmentados nos diferentes schemas e conformados na camada RAW através de procedimento que envolve:

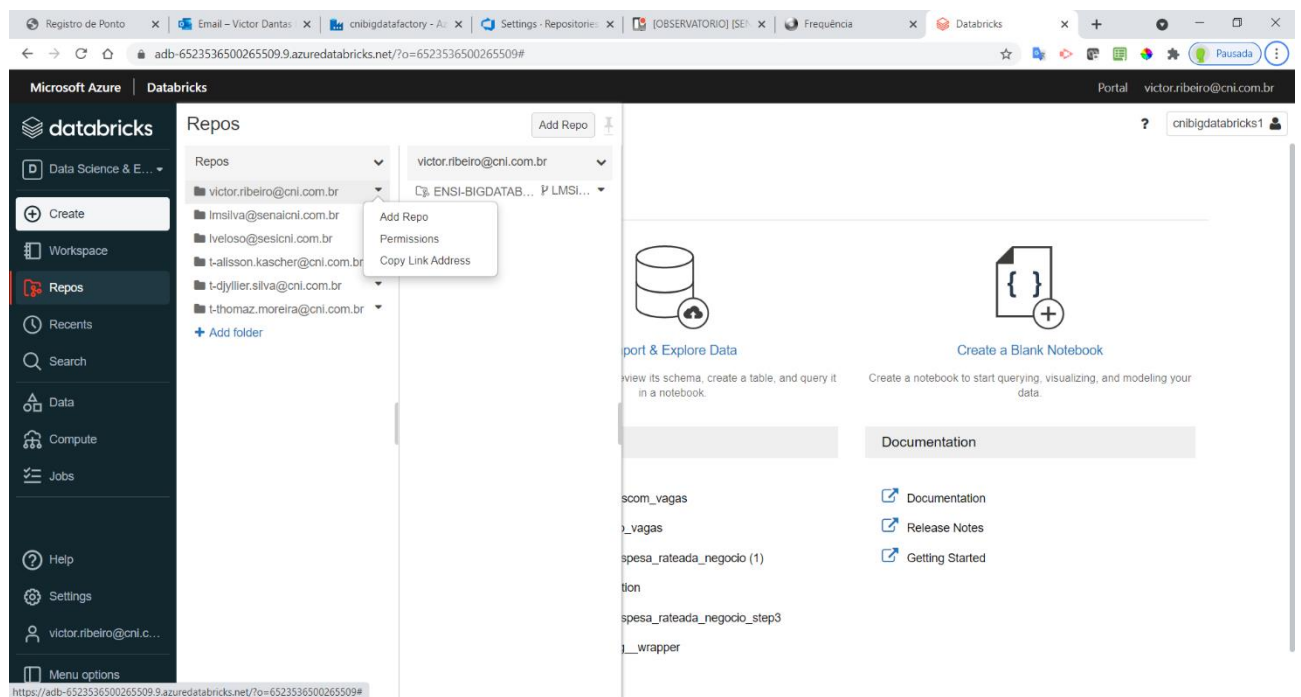
1 – Transformação dos dados via *Databricks*

2 – Orquestração e parametrização da transformação via *Azure Data Factory*.

5.1 – Camada RAW – Desenvolvimento com Azure Databricks

Para iniciar o desenvolvimento que qualquer ingestão de dados na camada RAW pelo Databricks os seguintes passos devem ser executados:

1. Clique na opção "Repos" no menu lateral esquerdo após fazer login na url <https://eastus2.azuredatabricks.net/>
2. Clique na opção "add repo" no diretório do seu e-mail de login, conforme a imagem:



3. Utilize a url https://dev.azure.com/CNI-STI/ENSI-BIG%20DATA/_git/ENSI-BIGDATABRI-CKS para fazer o clone do repositório.
4. Imediatamente crie uma ramificação conforme a documentação do link <https://docs.microsoft.com/pt-br/azure/databricks/repos> para poder visualizar, clonar e alterar notebook e abrir pull requests. **Observação:** Fica a critério do desenvolvedor a criação da branch para testes ou para efetivamente ser usada para realizar o pull request para a master, somente ressaltamos que o pull request para a master deve conter apenas os notebooks trabalhados pelo desenvolvedor na demanda.

5.2 – Camada RAW – Transformações com Azure Databricks

Os notebooks de transformação para a camada RAW são específicos para cada schema de dados. A versão de desenvolvimento, deve ser mantida no diretório */KEYRUS/dev/raw/crw/<nome da fonte>/<nome do processo>*.

Obrigatoriamente, os notebooks recebem 3 parâmetros: *table*, *adf* e *dls*. Abaixo, podemos compreender os parâmetros através do exemplo *inep_censo_escolar/org_raw_turmas*:

```
table = {
  "schema": "inep_censo_escolar",
  "table": "turmas",
  "partition_column_raw": "NU_ANO_CENSO",
  "prm_path":
"/prm/usr/inep_censo_escolar_turma/KC2332_Censo_Escolar_Turma_mapeamento_unificado_raw.xlsx"
}

adf = {
  "adf_factory_name": "cnibigdatafactory",
  "adf_pipeline_name": "org_raw_turmas",
  "adf_pipeline_run_id": "60ee3485-4a56-4ad1-99ae-6666666666",
  "adf_trigger_id": "62bee9e9-acbb-49cc-80f2-6666666666",
  "adf_trigger_name": "62bee9e9-acbb-49cc-80f2-6666666666",
  "adf_trigger_time": "2021-06-01 00:00:00",
  "adf_trigger_type": "PipelineActivity"
}

dls = {"folders": {
  "landing": "/lnd",
  "error": "/tmp/dev/err",
  "staging": "/tmp/dev/stg",
  "log": "/tmp/dev/log",
  "raw": "/raw"}
}
```

Vamos começar pela análise do parâmetro *table*:

- *schema*: é o nome da área de assunto na qual o dados estão persistidos na camada landing no data lake
- *table*: nome da tabela/estrutura de dados persistida no data lake. Geralmente, é utilizado em conjunto com o item anterior para construção do caminho de leitura (exemplo: `{Ind}/crw/{schema}__{table}`) e também do de destino dos dados (exemplo: `{adl_path}{raw}/crw/{schema}/{table}`) após o processamento
- *partition_column_raw*: nome da coluna de particionamento para escrita na camada *RAW*. O objeto é do tipo *string*. No caso de tabelas que serão particionadas por múltiplas colunas, declare um objeto do tipo array e trate o comportamento esperado dentro do notebook *Databricks*.
- *prm_path*: para transformações que necessitam de um dicionário de dados ou arquivos complementares, esses arquivos devem ser disponibilizados na área */prm* no data lake. O caminho para acesso a esse arquivo de parâmetros é informado por essa chave. Abrir, verificar, ler e formatar os dados do arquivo é uma responsabilidade do código do desenvolvedor, e, para isso, é possível utilizar bibliotecas *Python* e *Spark*. Para arquivos de parâmetros que seguem o padrão estabelecido pela STI-CNI, é possível fazer uso direto de uma biblioteca desenvolvida especificamente para facilitar o parsing dos arquivos de parâmetros, importada através do comando:

```
import crawler.functions as cf
```

O parâmetro *adf* traz os valores que serão diretamente inseridos na coluna *kv_process_control*, referenciada anteriormente neste documento, e que traz as informações sobre o processo de carga, permitindo a rastreabilidade completa do processo de carga. As chaves e valores são inseridos na tabela *RAW as is*.

O parâmetro *dls* é o mesmo parâmetro utilizado nos processos de carga de bases de dados OLTP (sistemas internos) e é gerado automaticamente pelo ADF, de acordo com a declaração da variável *env* no pipeline de carga. Com isso, os caminhos base dos arquivos, considerando origem e destino, são repassados ao notebook. O desenvolvedor deve utilizar esses caminhos base para construir o caminho completo.

5.2.1– Exemplo de uso de arquivos de parametrização

Como sabemos, muitas tabelas *RAW* de origem de fontes externas necessitarão de dicionários de dados para a correta conformação dos dados.

No diretório de anexos desse documento, o arquivo *3.2.7.1.1_exemplo_raw_crw_parametros* traz o exemplo de uma implementação que utiliza um arquivo de parâmetros nos padrões STI-CNI e a biblioteca de *parsing* dos parâmetros, mencionada anteriormente.

Neste exemplo, ainda se faz o uso de um laço iterativo para a leitura de cada arquivo persistido pelo *bot* na camada *landing*. O trecho de código é:

```
years = list(map(lambda path: path.split('/')[-1],
cf.list_subdirectory(dbutils,'{Ind}/crw/{schema}__{table}'.format(Ind=Ind,
schema=table['schema'],table=table['table']))))

prm_years = sorted(var_prm_dict.keys())
```

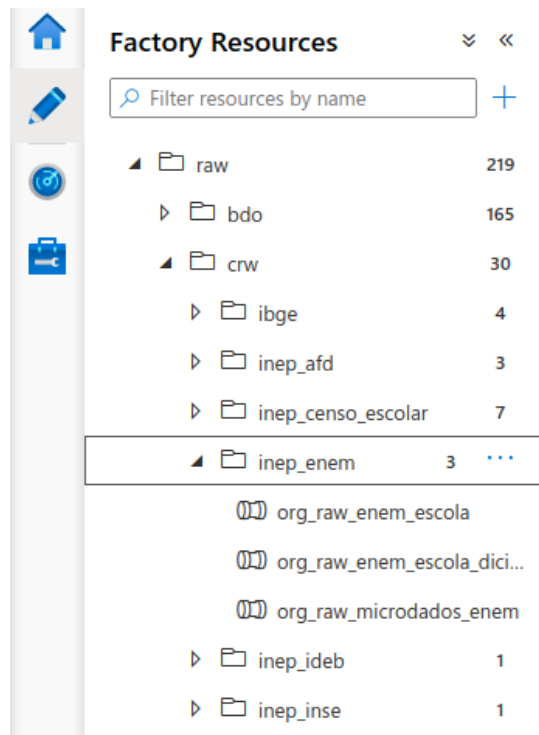
Muitas transformações *RAW* fazem uso desse tipo de recurso para mapear os arquivos capturados para as partições de escrita na camada *RAW*. A necessidade do uso desse tipo de recurso deve ser avaliada para cada caso. Há casos documentados no projeto que é necessário iterar por ano e mês. O desenvolvedor deve se atentar a essas necessidades e prover uma estratégia eficiente de implementação.

5.3 – Camada RAW – Orquestração com Azure Data Factory

Paralelamente ao desenvolvimento das transformações no *Azure Databricks*, é necessário desenvolver, no *Azure Data Factory*, o pipeline para agendamento e orquestração da carga dos dados.

Numa nova *branch*, seguindo os padrões de nomenclatura definidos pela STI-CNI, um novo pipeline deve ser criado em:

raw/crw/<área de assunto do bot>/org_raw_<tabela>



O novo pipeline deve conter 3 parâmetros.

- **tables:**

Tipo: *object*

Descrição: mesmas chaves e valores definidos para o objeto tables na seção 3.2.7.1

Exemplo:

```
{
  "schema": "inep_censo_escolar",
  "table": "turmas",
  "partition_column_raw": "NU_ANO_CENSO",
  "prm_path":
    "/prm/usr/inep_censo_escolar_turma/KC2332_Censo_Escolar_Turma_mapeamento_unificado_raw.xlsx"
}
```

- **databricks:**

Tipo: *object*

Descrição: Deve conter, minimamente a chave notebook, para a qual o valor deve ser o caminho relativo ao notebook que será executado para a transformação dos dados. O

caminho deve desconsiderar os dois primeiros níveis dentro do *Databricks Workspace*, ou seja, não é necessário incluir */KEYRUS/dev* e */KEYRUS/prod*.

Exemplo:

```
{"notebook":"/inep_afd/org_raw_afd_escolas"}
```

- **env:**

Tipo: *object*

Descrição: dicionário com a declaração do ambiente destino da execução.

Exemplo:

```
{"env": "dev"}
```

O pipeline deve implementar uma única atividade do tipo *Execute Pipeline*, que faz chamada ao template *import_crw__0__switch_env*. Como parâmetros, deve prover o seguinte mapeamento, utilizando contexto dinâmico para todos os casos:

```
env = @pipeline().parameters.env

adf = {'adf_factory_name': '@{pipeline().DataFactory}', 'adf_pipeline_name':
'@{pipeline().Pipeline}', 'adf_pipeline_run_id': '@{pipeline().RunId}', 'adf_trigger_id':
'@{pipeline().TriggerId}', 'adf_trigger_name': '@{pipeline().TriggerName}',
'adf_trigger_time': '@{pipeline().TriggerTime}', 'adf_trigger_type':
'@{pipeline().TriggerType}'}

tables = @pipeline().parameters.tables

databricks = @pipeline().parameters.databricks

ach_tables = {"null": "null"}
```

O parâmetro *ach_tables* é declarado *{"null": "null"}* para os casos nos quais os arquivos originais não precisam ser armazenados na área *archive (ach)*. Para os casos em que esse comportamento é esperado, o parâmetro do tipo *string*, assume a representação de uma sequência de dicionários, separados por ponto e vírgula, para o mapeamento dos arquivos para suas respectivas áreas em *ach*. Tomemos como exemplo a implementação em *org_raw_ideb_ideb*:

```
ach_tables =
{'schema':'inep_ideb','table':'brasil_ai'};{'schema':'inep_ideb','table':'brasil_em'};{'sche
ma':'inep_ideb','table':'escolas_af'};{'schema':'inep_ideb','table':'escolas_ai'};{'schema'
:'inep_ideb','table':'escolas_em'};{'schema':'inep_ideb','table':'estados_regioes_af'};{'sc
hema':'inep_ideb','table':'estados_regioes_ai'};{'schema':'inep_ideb','table':'estados_re
gioes_em'};{'schema':'inep_ideb','table':'municipios_af'};{'schema':'inep_ideb','table':'
municipios_ai'};{'schema':'inep_ideb','table':'municipios_em'}
```

Nas opções gerais do pipeline, em Settings, defina a *Concurrency* = 1.

Na aba de propriedades do pipeline, adicione uma descrição para descrever brevemente o processo.

Em seguida, adicione as anotações: *raw*, *crawler*, <área de assunto>

No data lake, esse mapeamento se reflete na seguinte estrutura em */ach/crw/inep_ideb*:

Authentication method: Access key (Switch to Azure AD User Account)
Location: datalake / ach / crw / inep_ideb













Search blobs by prefix (case-sensitive)

	Name	Modified	Access
<input type="checkbox"/>	📁 [..]		
<input type="checkbox"/>	📁 brasil_af		
<input type="checkbox"/>	📁 brasil_ai		
<input type="checkbox"/>	📁 brasil_em		
<input type="checkbox"/>	📁 escolas_af		
<input type="checkbox"/>	📁 escolas_ai		
<input type="checkbox"/>	📁 escolas_em		
<input type="checkbox"/>	📁 estados_regioes_af		
<input type="checkbox"/>	📁 estados_regioes_ai		
<input type="checkbox"/>	📁 estados_regioes_em		
<input type="checkbox"/>	📁 municipios_af		
<input type="checkbox"/>	📁 municipios_ai		

Dentro de cada diretório com o nome definido pela chave *tables*, encontramos diretórios com o timestamp de execução do processo. Nesses diretórios estão os arquivos originais, coletados pelos *bots*.

Authentication method: Access key (Switch to Azure AD User Account)
Location: datalake / ach / crw / inep_ideb / brasil_af

Search blobs by prefix (case-sensitive)

Name	Modified
<input type="checkbox"/>  [..]	
<input type="checkbox"/>  202101061412468777892	
<input type="checkbox"/>  202102041257097679794	
<input type="checkbox"/>  202102041406278738552	
<input type="checkbox"/>  202102041610555433182	
<input type="checkbox"/>  202102041743243882232	
<input type="checkbox"/>  202102042057295647427	
<input type="checkbox"/>  20210205125021685201	
<input type="checkbox"/>  202102051609534593802	
<input type="checkbox"/>  202102051649507866111	
<input type="checkbox"/>  202102052052364175071	
<input type="checkbox"/>  202102081312436252305	

Após esses passos, o pipeline pode ser testado através do botão *Debug*. O procedimento para publicação em produção segue as mesmas diretrizes definidas na seção 3.2.6. Todos os objetos de desenvolvimento no *Data Factory* seguem a estratégia de *Pull Request* para publicação.

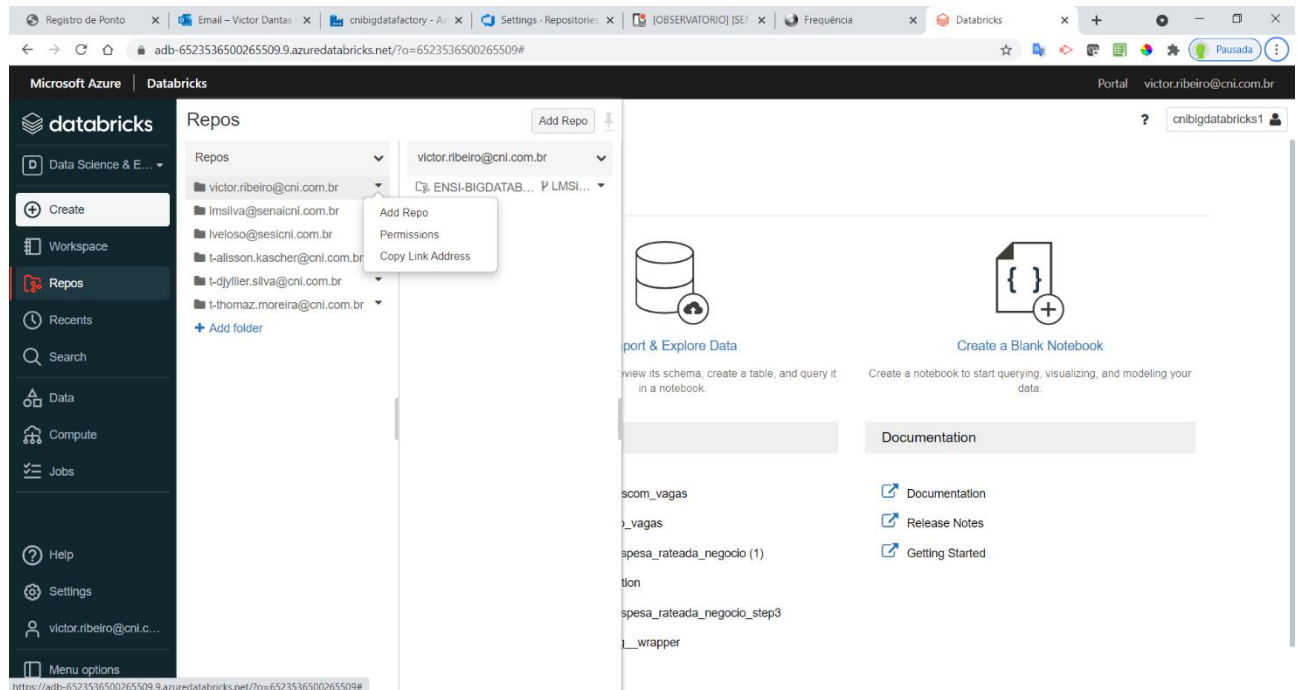
6. Padrões de desenvolvimento – camada TRS

6.1 Desenvolvimento no Azure Databricks

Se ainda não realizou o mapeamento do repositório do projeto, execute os seguintes passo antes de iniciar:

1. Clique na opção "Repos" no menu lateral esquerdo após fazer login na url <https://eastus2.azuredatabricks.net/>

2. Clique na opção "add repo" no diretório do seu e-mail de login, conforme a imagem:

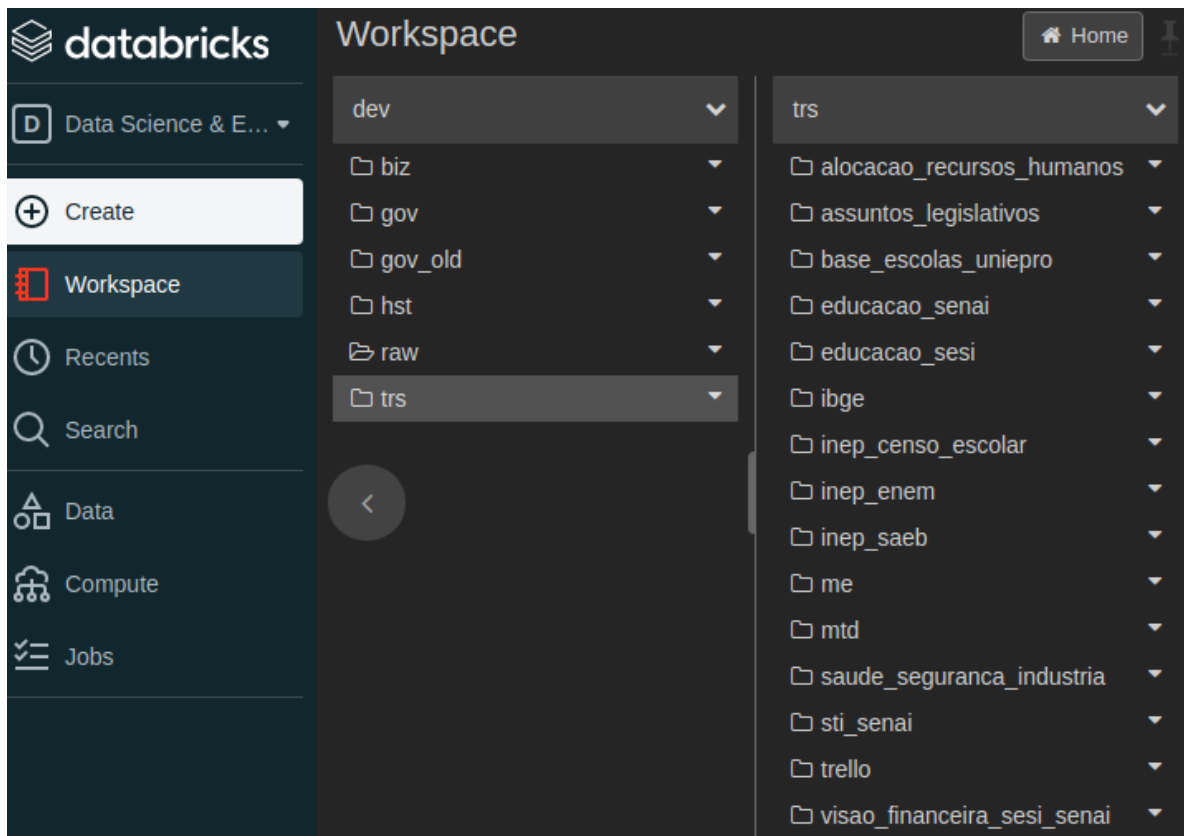


3. Utilize a url https://dev.azure.com/CNI-STI/ENSI-BIG%20DATA/_git/ENSI-BIGDATABRICKS para fazer o clone do repositório.
4. Imediatamente crie uma ramificação conforme a documentação do link <https://docs.microsoft.com/pt-br/azure/databricks/repos> para poder visualizar, clonar e alterar notebook e abrir pull requests. **Observação:** Fica a critério do desenvolvedor a criação da branch para testes ou para efetivamente ser usada para realizar o pull request para a master, somente ressaltamos que o pull request para a master deve conter apenas os notebooks trabalhados pelo desenvolvedor na demanda.

Para a camada *trusted* (TRS), não há diferenciação do procedimento de acordo com a origem. Sistemas OLTP e *crawlers* não são objetos de interesse destas transformações, que consideram apenas a área *raw*, camada que apresenta conformação tabular e possui governança de conteúdo e acesso.

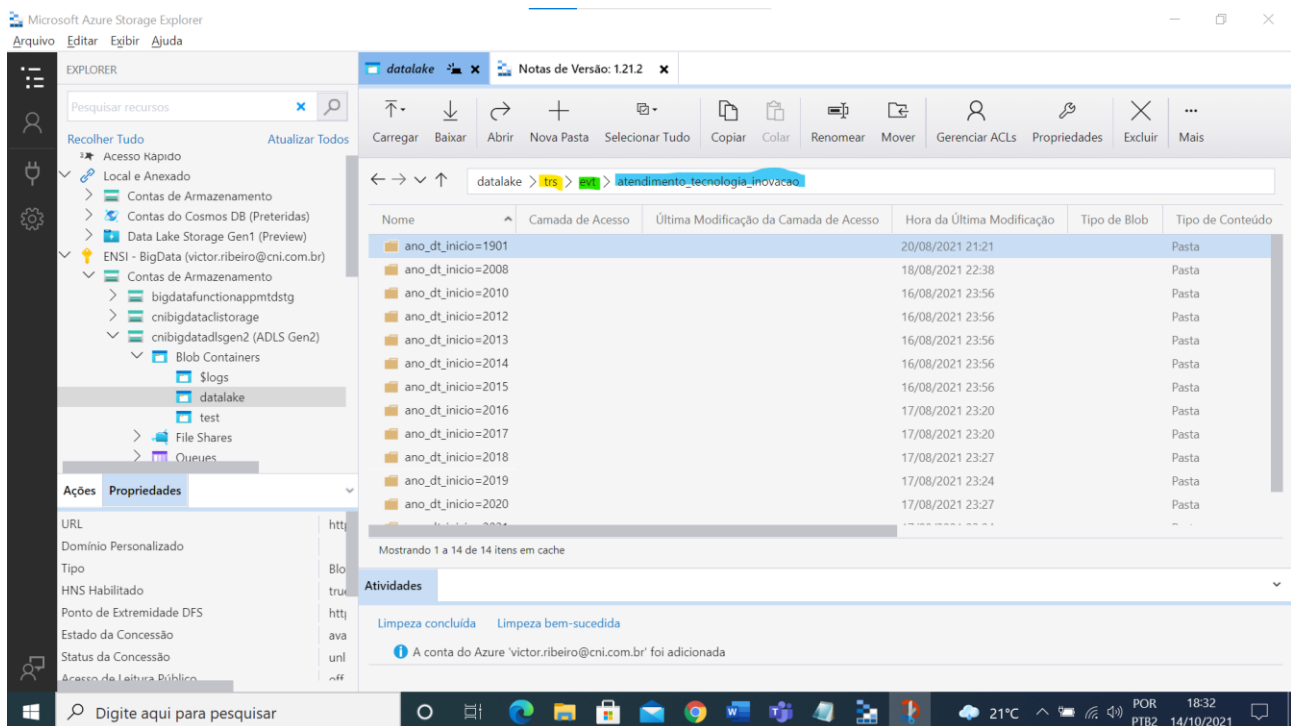
Tabelas *trusted* abstraem uma visão validada dos dados e trazem garantias quanto à granularidade, singularidade, formatação e usabilidade para a composição de objetos que indicam melhor a visão do negócio. Um objeto *trusted* pode ter uma ou mais tabelas *raw* como origem.

No Databricks, as tabelas *trs* são segmentadas por área de assunto dentro do *workspace*, e todas as tabelas possuem uma única área de assunto – definida pela equipe de analistas de negócio e especificada na documentação de desenvolvimento.



Como na camada *raw*, todos os objetos são identificados por um nome de processo, que também é definido pela equipe de análise de requisitos de negócio.

Sempre um processo/notebook de camada Trusted deve escrever os dados no **caminho** **/trs/<assunto>/<table>**, **respeitando 3 níveis de diretórios**, pois o padrão de desenvolvimento vai buscar essa estrutura se os parâmetros forem preenchidos de forma correta. Veja os exemplos da imagem a seguir: Amarelo=caminho base, verde=assunto, azul=table



Quanto às transformações, elas são específicas para a exata visão que deve ser gerada. Embora não haja um template unificado para gerar as tabelas desse contexto, há parâmetros e estratégias de persistência padronizados que cabem ser discutidos neste documento. Vamos começar pelos parâmetros.

Os notebooks, em geral, esperam 3 parâmetros:

- *tables*: dicionários com os caminhos relativos das origens (para leitura) e do destino (para escrita) da tabela *trusted* em questão. As origens podem ser múltiplas (*array*) e o destino, por ser uma única tabela, é um valor simples (*string*). Quando há múltiplas origens, o desenvolvedor é responsável por definir a ordem de leitura do vetor e como mapeá-lo para os *Dataframes*.

Exemplo:

```
{
  "origins": ["/bdo/bd_basi/rl_atendimento_metrica", "/bdo/bd_basi/tb_atendimento"],
  "destination": "/evt/convenio_ensino_prof_carga_horaria"
}
```

- *dls*: dicionário com os caminhos base do data lake que apontam para a seção lógica do ambiente definido no parâmetro do ADF. Semelhante às outras ocorrências já discutidas nesse documento.

Exemplo:

```
{
  "folders":{
    "landing":"/tmp/dev/ln",
    "error":"/tmp/dev/err",
    "staging":"/tmp/dev/stg",
    "log":"/tmp/dev/log",
    "raw":"/tmp/dev/raw",
    "trusted": "/tmp/dev/trs"
  }
}
```

- *adf*: dicionário com os identificadores de execução do ADF; usado para popular a coluna *kv_process_control*, já apresentado anteriormente no documento.

Exemplo:

```
{
  "adf_factory_name": "cnibigdatafactory",
  "adf_pipeline_name": "raw_trs_convenio_ensino_prof_carga_horaria",
  "adf_pipeline_run_id": "62bee9e9-acbb-49cc-80f2-66666666",
  "adf_trigger_id": "80eae9e9-acbb-49cc-80f2-45cad311",
  "adf_trigger_name": "60f45ad-acbb-49cc-80f2-ccd667a",
  "adf_trigger_time": "2020-05-26T17:57:06.0829994Z",
  "adf_trigger_type": "Manual"
}
```

Novamente, esses parâmetros devem ser definidos no ADF e repassados ao Databricks como *widgets*, da mesma forma como na camada *raw*. Para a aplicação das **transformações**, algumas sugestões devem ser observadas:

1 - Evite o uso de %SQL. Embora pareça mais fácil desenvolver em SQL, o plano de execução gerado automaticamente pelo *Databricks* não traz os finos ajustes de um desenvolvedor experiente, além de requerer a escrita de tabelas intermediárias na área gerenciada pelo *HiveMetastore* no caso de trabalhos que necessitam de ser segmentados em etapas.

2 – Não utilize o Databricks *DataStore/MetaStore* para escrita de dados, nem de objetos temporários. Transformações bem desenvolvidas não necessitam de áreas *stage* (tem se mantido verdadeiro até o momento). Caso seja imprescindível a implementação de uma área *stage* no

DataStore, a transformação deve cuidar da sua manutenção e gerenciamento. Implementações que utilizam essa estratégia devem ser informadas à equipe STI-CNI e devem ser aprovadas para que possam seguir.

3 – Utilize *cache()* e *unpersist()*. O controle manual (e inteligente) de caches permite acelerar os processos de reduzir o tráfego desnecessário de dados entre o data lake e o *Databricks*.

4 – Utilize *delta* ou *parquet* para a escrita. Os dois formatos são permitidos. No formato *parquet*, não há a possibilidade de realizar *updates* e *deletes* diretamente. A lógica para permitir esses comportamentos se baseia em teoria de conjuntos e sobrescrita de partição. Apenas usuários experientes devem utilizar esse método. Para o formato *delta*, lembre-se de aplicar *vacuum()* e *optimize()*, nas tabelas.

5 – Remova *count()*, *show()*, e quaisquer outras ações desnecessárias para a execução da transformação antes de entregar seu notebook para homologação. Esses comandos disparam *Spark Jobs* e oneram o processamento apenas para mostrar informações que só tem utilidade em tempo de desenvolvimento. Após sua conferência, devem ser comentados ou removidos do código.

Para os novos objetos desenvolvidos nessa camada, recomendamos o uso do formato *delta*, por permitir transações *ACID* e operações que outrora eram de grande dificuldade de implementação. Para as operações de *update*, *delete* ou *merge* utilizando *delta*, utilize essa referência:

<https://docs.databricks.com/delta/delta-update.html#language-python>

Aos que permanecerão desenvolvendo com o formato *parquet*, utilizem as referências dos anexos a seguir para os comportamentos de carga listados abaixo:

- **Full:** 4.1_exemplo_raw_trs_full.html
- **Append:** 4.1_exemplo_raw_trs_append.html
- **Update:** 4.1_exemplo_raw_trs_update.html
- **Versioning:** 4.1_exemplo_raw_trs_versioning.html

Cada exemplo traz especificidades quanto à comparação de modificações dos registros assim como a estratégia de sobrescrita de partição. É necessário estudar cada caso e compreender os passos necessários à implementação. Novamente, recomendamos essa estratégia apenas para desenvolvedores experientes.

6.2 Desenvolvimento no Azure Data Factory

No ADF, é necessário desenvolver o pipeline de agendamento/coordenação da carga. Os objetos devem ser criados no diretório da área de assunto, dentro de *trs* e seguem uma nomenclatura já conhecida:

trs/<área>/raw_trs_<nome do processo>

O pipeline deve conter quatro parâmetros:

- **tables:**

Tipo: *object*

Descrição: dicionário com o mapeamento de origens e destino do processamento, exatamente como apresentado na seção 4.1.

Exemplo:

```
{
  "origins": ["/bdo/bd_basi/rl_atendimento_metrica",
    "/bdo/bd_basi/tb_atendimento"],
  "destination": "/evt/convenio_ensino_prof_carga_horaria"
}
```

- **databricks:**

Tipo: *object*

Descrição: dicionário com o mapeamento do caminho relativo do notebook que será executado pelo *Databricks*. Os parâmetros contêm a chave mais externa *trs* e na estrutura interna dessa chave, o para *notebook:<caminho relativo>*, que, como de costume, despreza os níveis do *workspace* até o nível do ambiente.

Exemplo:

```
{
  "trs": {
    "notebook": "/trs/educacao_senai/raw_trs_convenio_ensino_profissional"
  }
}
```

```
}  
}
```

- ***user_parameters***:

Tipo: *object*

Descrição: dicionário genérico que pode ser usado para envio de parâmetros adicionais para a execução dos notebooks. Não há uma regra para preenchimento deste objeto. O intuito de seu uso é permitir customizações da execução dos notebooks não previstas pelo framework generalizado, é um objeto que proporciona extensibilidade das funcionalidades. Para os casos comuns, deve ser preenchido com:

```
{"null": "null"}
```

- ***env***:

Tipo: *object*

Descrição: dicionário com a chave *env* que informa o ambiente para execução do processo.

Exemplo:

```
{"env": "dev"}
```

O pipeline deve implementar uma única atividade do tipo *Execute Pipeline* que aponta para o *template trusted__0__switch_env*. Nas configurações da atividade, os seguintes valores devem ser preenchidos com contexto dinâmico:

```
env = @pipeline().parameters.env  
tables= @pipeline().parameters.tables  
databricks = @pipeline().parameters.databricks  
user_params = @pipeline().parameters.user_params  
adf = {'adf_factory_name': '@{pipeline().DataFactory}', 'adf_pipeline_name':  
'@{pipeline().Pipeline}', 'adf_pipeline_run_id': '@{pipeline().RunId}', 'adf_trigger_id':  
'@{pipeline().TriggerId}', 'adf_trigger_name': '@{pipeline().TriggerName}',  
'adf_trigger_time': '@{pipeline().TriggerTime}', 'adf_trigger_type':  
'@{pipeline().TriggerType}'}
```

Saved Save as template Validate Debug Add trigger

Execute Pipeline trusted_0_switch_env

General Settings User properties

Invoked pipeline * trusted_0_switch_env Open New

Wait on completion ☒

Parameters

Expand toolbox pane

	Type	Value	Default value
env	object	@pipeline().parameters.env	{"env": "dev"}
tables	object	@pipeline().parameters.tables	
databricks	object	@pipeline().parameters.databricks	
user_params	object	Value	{"null": "null"}
adf	string	{'adf_factory_name': '@{pipeline().Data...	

Nas opções gerais do pipeline, em Settings, defina a *Concurrency* = 1.

Na aba de propriedades do pipeline, adicione uma descrição para explicar brevemente o processo. Em seguida, adicione as anotações: *trusted*, <área de assunto>

Para os testes de execução, basta clicar no botão Debug e seguir o procedimento de publicação em produção via *Pull Requests* descrito anteriormente nesse documento.

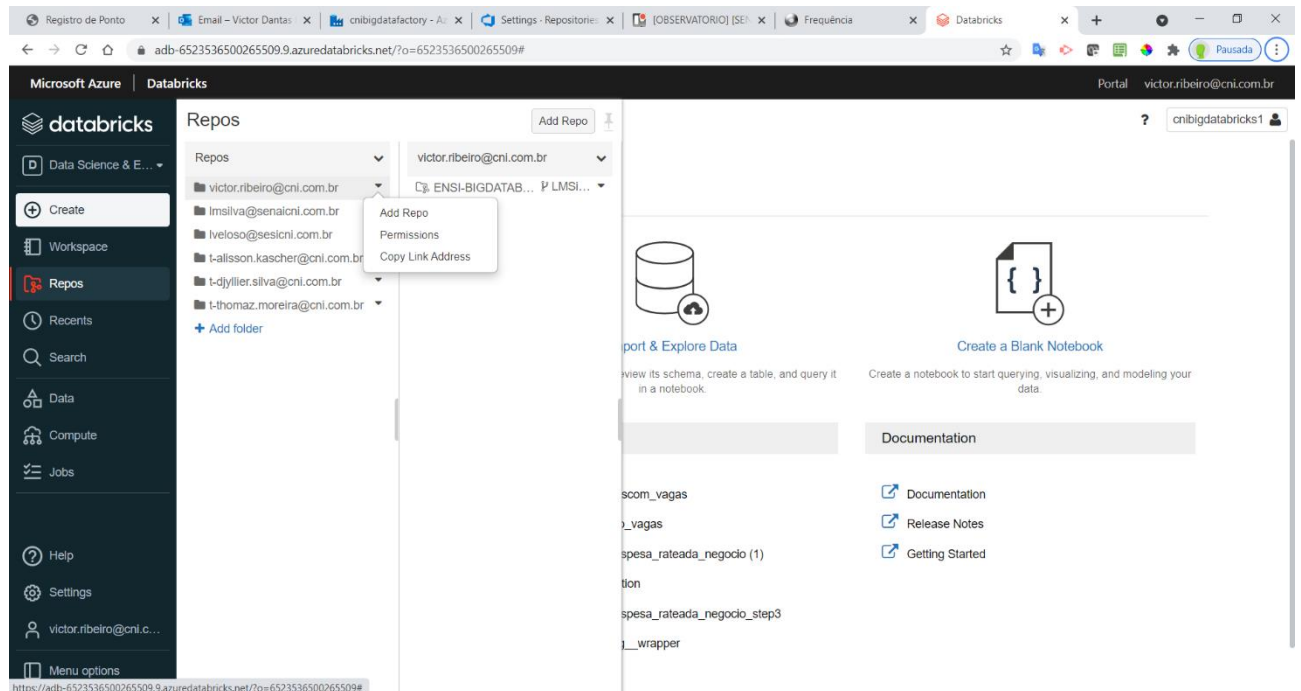
7. Padrões de desenvolvimento – camada BIZ

7.1 – Desenvolvimento no Azure Databricks

Se ainda não realizou o mapeamento do repositório do projeto, execute os seguintes passo antes de iniciar:

1. Clique na opção "Repos" no menu lateral esquerdo após fazer login na url <https://eastus2.azuredatabricks.net/>

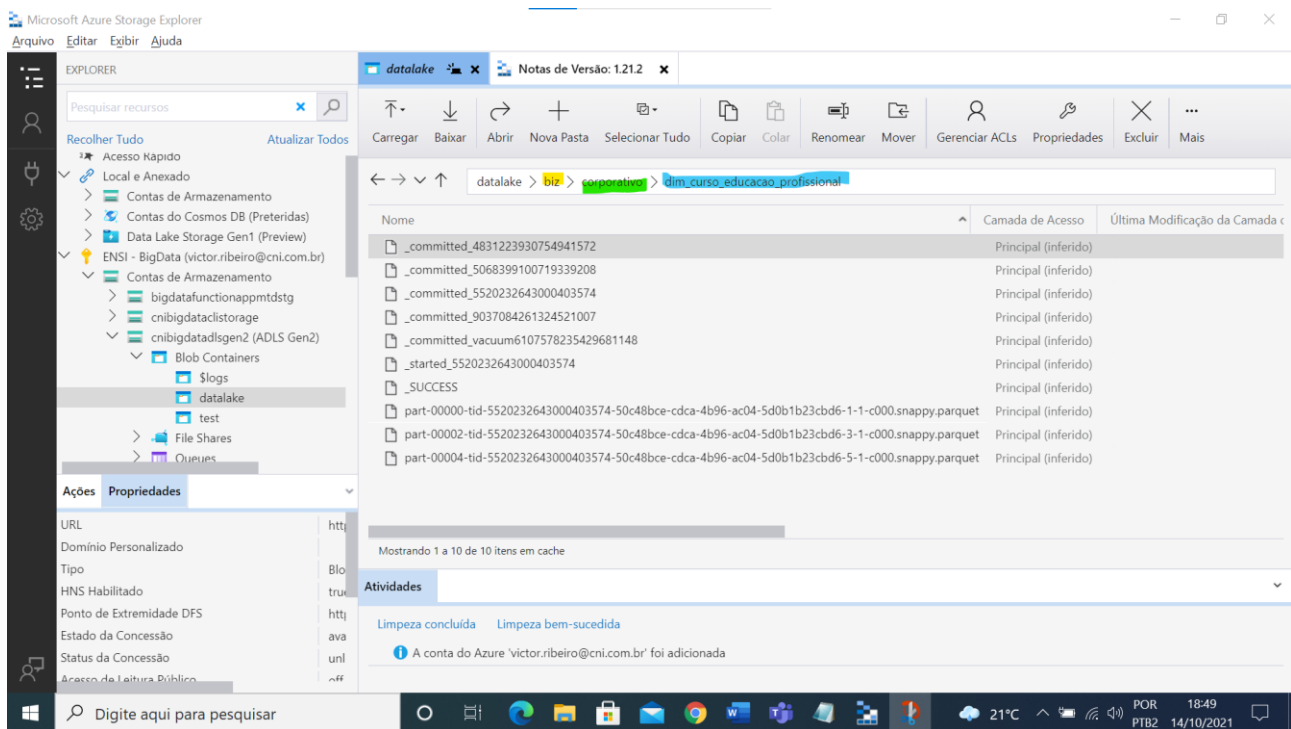
2. Clique na opção "add repo" no diretório do seu e-mail de login, conforme a imagem:



3. Utilize a url https://dev.azure.com/CNI-STI/ENSI-BIG%20DATA/_git/ENSI-BIGDATABRICKS para fazer o clone do repositório.
4. Imediatamente crie uma ramificação conforme a documentação do link <https://docs.microsoft.com/pt-br/azure/databricks/repos> para poder visualizar, clonar e alterar notebook e abrir pull requests. **Observação:** Fica a critério do desenvolvedor a criação da branch para testes ou para efetivamente ser usada para realizar o pull request para a master, somente ressaltamos que o pull request para a master deve conter apenas os notebooks trabalhados pelo desenvolvedor na demanda.

Para os notebooks de transformação da camada *biz*, as mesmas observações apresentadas no item 4.2 podem ser observadas. Os notebooks aqui também são específicos para cada visão individualmente especificada.

Sempre um processo/notebook de camada Business deve escrever os dados no **caminho /biz/<área de assunto da biz>/<table>**, **respeitando 3 níveis de diretórios**, pois o padrão de desenvolvimento vai buscar essa estrutura se os parâmetros forem preenchidos de forma correta. Veja os exemplos da imagem a seguir: Amarelo=caminho base, verde=assunto, azul=table



Nessa camada, os notebooks recebem, por padrão, quatro parâmetros. Dentre eles, os parâmetros *adf* e *dls* já foram apresentados diversas vezes nesse documento (rever seção 4.2) e, por questão de simplificação, vamos nos ater aos dois novos parâmetros:

- **tables:**

Tipo: *object*

Descrição: dicionário com os caminhos relativos das origens, do destino no data lake e do notebook *Databricks* dentro do *workspace*. Os valores relacionados ao *Databricks* não são usados dentro do notebook, mas servem para o mapeamento da tarefa no nível do *ADF*. Novamente, as origens podem ser múltiplas (*array*) e devem ser objetos da camada *trusted*. O destino é único, já que cada notebook implementa uma única tabela.

Exemplo:

```
{
  "origins": ["/evt/convenio_ensino_profissional",
              "/evt/convenio_ensino_prof_carga_horaria"
            ],
  "destination": "/producao/fta_convenio_ensino_profissional",
  "databricks": {
```

```

        "notebook":
        "/biz/educacao_senai/trs_biz_fta_convenio_ensino_profissional"
    }
}

```

- *user_parameters*:

Tipo: *object*

Descrição: dicionário utilizado para trazer instruções adicionais e customizadas para uso nos notebooks. Para itens que implementam fechamento, esse dicionário traz a chave *closing*, que no seu valor especifica os parâmetros de fechamento: *year*, *month* e *dt_closing*. Para os meses de fechamento, não utilize dois dígitos para os meses menores que 10. O mês de janeiro, por exemplo, deve ser informado como *month=1*.

Para os casos em que não há fechamento, geralmente esse parâmetro assume o valor {"null": "null"}.

Exemplo:

```

{
    "closing": {
        "year": 2019,
        "month": 12,
        "dt_closing": "2020-03-02"
    }
}

```

Para a maior parte dos casos de fechamento, quando não são declarados os parâmetros de fechamento – ou seja, quando a *closing* está nas chaves declaradas em *user_parameters* – o processo deve assumir a data atual de processamento para o fechamento. Nos notebooks que devem utilizar essa regra, o seguinte trecho de código pode ser encontrado:

```

var_parameters = {}
if "closing" in var_user_parameters:
    if "year" and "month" and "dt_closing" in var_user_parameters["closing"] :

```

```

var_parameters["prm_ano_fechamento"] = var_user_parameters["closing"]["year"]
var_parameters["prm_mes_fechamento"] =
var_user_parameters["closing"]["month"]
splited_date = var_user_parameters["closing"]["dt_closing"].split('-', 2)
var_parameters["prm_data_corte"] = date(int(splited_date[0]), int(splited_date[1]),
int(splited_date[2]))
else:
var_parameters["prm_ano_fechamento"] = datetime.now().year
var_parameters["prm_mes_fechamento"] = datetime.now().month
var_parameters["prm_data_corte"] = datetime.now()

print(var_parameters)

```

Como esse trecho de código tem sido utilizado com bastante frequência, em breve deve ser absorvido por uma biblioteca para distribuição a nível corporativo. De toda forma, é uma lógica bem simples e pode ser acrescentada explicitamente ao código sempre que necessário.

As tabelas *biz* implementam o fechamento através da filtragem dos registros que são iguais ou inferiores aos parâmetros em *closing*. Após essa operação, no final, toda a tabela é sobrescrita obedecendo aos dados filtrados.

As transformações dessa camada costumam ser mais simples se comparadas à *trusted*. Muito pode ser reaproveitado entre as implementações. Um bom exemplo, considerando fechamento, pode ser encontrado no anexo *5.1_exemplo_trs_biz_closing.html*.

7.2 Desenvolvimento no Azure Data Factory

No ADF, também será necessário um pipeline para orquestração desse processo, que deve ser persistido em:

`biz/<área de assunto da biz>/trz_biz_<processo>`

O pipeline necessita de três parâmetros:

- tables:

Tipo: object

Descrição:

- user_parameters:

Tipo: object

Descrição:

- env:

Tipo: object

Descrição:

8. Padrões de desenvolvimento- Workflows

8.1 Desenvolvimento dos Workflows no Azure Data Factory

Após as etapas anteriores, o desenvolvedor deve orquestrar seu workflow responsável por rodar os pipelines por ele desenvolvido. Este workflow fará parte de uma estrutura maior executada pelo workflow_prod, para os dados que serão escritos nas camadas raw, trs, biz. Já os pipelines de execução dos bots e escrita na camada lnd, são executados pelo workflow prod_lnd.

No Adf, os workflows devem ser persistidos na pasta “workflow”.

▲	📁 workflow	79
	📄 workflow	
	📄 workflow_dev	
	📄 workflow_planilhas_prod	
	📄 workflow_prod	
	📄 workflow_prod_lnd	
	📄 workflow_prod_raw_uld	
▶	📁 assuntos_legislativos	2
▶	📁 educacao_senai	4
▶	📁 educacao_sesi	4
▶	📁 fechamentos	5
▶	📁 fechamentos_wrapper	5
▶	📁 gov	8

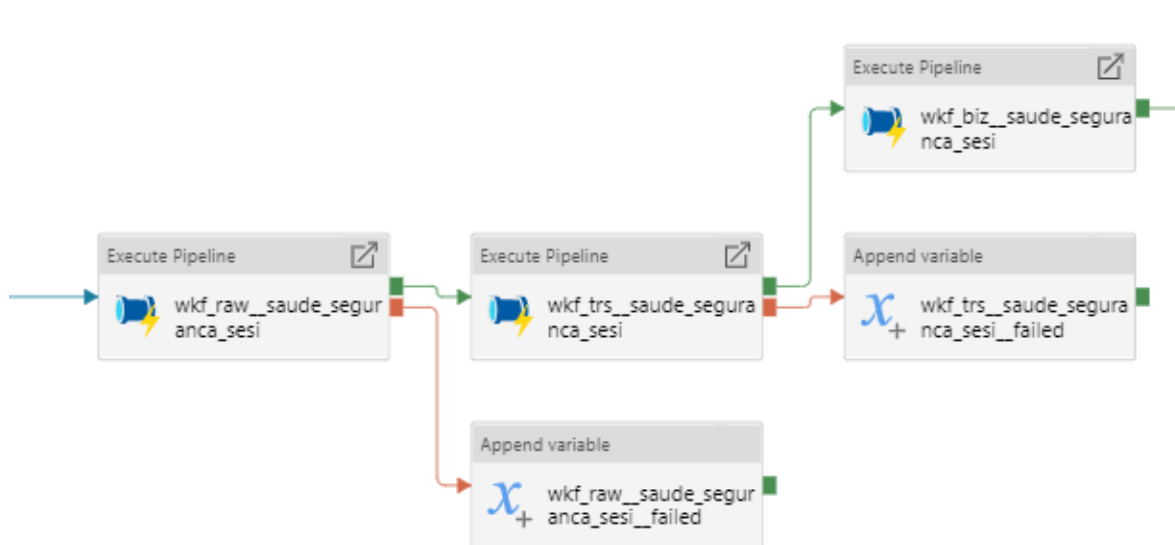
Exemplos de referência

workflow: workflow agendado para executar as cargas das camadas raw, trusted e business da área de negócio UNIGEST(sistemas SCAE, SMD, PROTHEUS e etc).

workflow_prod_Ind: workflow que executa pipelines desenvolvidos para a camada landing(Ind)

workflow_prod__raw_uld: Executa os pipelines da camada raw e uld da área de negócio UNIEPRO e por necessidade de possuir um agendamento distinto da Ind foram criados workflows separados

O workflow de prod executa os workflows de acordo com uma sequência lógica, começando pela camada raw, em seguida trusted e enfim business.



Por fim, somente os workflows são executados com agendamento e este agendamento é configurado e proposto pelo fornecedor via pull request no Azure DevOps e a equipe da STI realiza análise das janelas e a aprovação quando for o caso.

9. Controle de versão

Versão	Data	Principais Alterações	Responsável
1.0	13/05/21	Versão inicial, conceitos e nomenclaturas.	Cecília Macedo
1.0	01/06/21	Inclusão dos padrões de desenvolvimento.	Cecília Macedo
2.0	14/06/21	Inclusão de novo conteúdo (Transformações TRS e BIZ).	Thomaz Moreira
2.1	18/08/21	Revisão e ajuste de alguns trechos de texto. Formatação.	Thiago Silva
2.2	19/08/21	Revisão e ajuste de alguns trechos de texto. Formatação.	Leonardo Mafra
2.3	20/08/21	Revisão do cofre de senhas.	Victor Dantas
2.4	20/08/21	Revisão geral do documento	Cecília Macedo
2.5	14/10/21	Revisão geral do documento	Victor Dantas
2.6	24/12/21	Inclusão de novo conteúdo (mensagem de retorno crawlers)	Victor Dantas
2.7	27/01/22	Inclusão de novo conteúdo (melhorias com base na experiência de uso do manual pelo Felipe Langoni)	Victor Dantas
2.8	31/03/22	Inclusão de novo conteúdo (Estrutura de diretórios do ADF e desenvolvimento dos workflows)	Felipe Langoni
3.0	31/03/22	Revisão geral do documento, reestruturação do índice, inclusão do conteúdo referente ao desenvolvimento de bots em DEV	Felipe Langoni e Victor Ribeiro