

O que iremos aprender

Na aula de hoje iremos explorar como realizar validações com great expectations em um projeto com esquema de arquivos

A ideia é explorarmos um cenário mais próximo do produtivo, ao invés de utilizarmos o jupyter notebook.

Hands on

Instalando o Great expectations

Bora lá então galera, o primeiro passo é se certificar que o Great Expectations está instalado em seu ambiente Python. Você pode fazer isso executando os seguintes comandos no seu terminal:

```
pip install great_expectations
great_expectations --version
```

Inicializando o projeto

Após executar esse comando, você pode executar o comando para inicializar um projeto do great expectations:

great_expectations init

```
Great Expectations
~ Always know what to expect from your data ~

Let's create a new Data Context to hold your project configuration.

Great Expectations will create a new directory with the following structure:

great_expectations
|-- great_expectations.yml
|-- expectations
|-- checkpoints
|-- plugins
|-- .gitignore
|-- uncommitted
    |-- config_variables.yml
    |-- data_docs
    |-- validations

OK to proceed? [Y/n]: Y

Congratulations! You are now ready to customize your Great Expectations configuration.

You can customize your configuration in many ways. Here are some examples:

Use the CLI to:
- Run 'great_expectations datasource new' to connect to your data.
- Run 'great_expectations checkpoint new <checkpoint_name>' to bundle data with Expectation Suite(s) in a Checkpoint for later re-validation.
- Run 'great_expectations suite --help' to create, edit, list, profile Expectation Suites.
- Run 'great_expectations docs --help' to build and manage Data Docs sites.

Edit your configuration in great_expectations.yml to:
- Move Stones to the cloud
- Add Slack notifications, PagerDuty alerts, etc.
- Customize your Data Docs

Please see our documentation for more configuration options!
```

A estrutura do projeto

great_expectations

-- great_expectations.yml # Este é um arquivo YAML que contém a configuração principal do projeto Great Expectations. Ele geralmente inclui informações como conexões de banco de dados, configurações de armazenamento e outras configurações de projeto.

-- expectations # Este é um diretório onde você pode armazenar os arquivos de especificações de expectativas. As expectativas são as regras ou condições que seus dados devem atender.

```
|-- checkpoints # Este diretório pode conter checkpoints, que são conjuntos de expectativas
que foram validadas em um determinado momento no tempo. Eles podem ser úteis para rastrear
a qualidade dos dados ao longo do tempo.

|-- plugins # Este diretório pode conter plugins adicionais para o Great Expectations. Os
plugins podem adicionar funcionalidades extras, como suporte a diferentes tipos de fontes de
dados ou integrações com outras ferramentas.

|-- .gitignore # Este é um arquivo que especifica quais arquivos e diretórios devem ser
ignorados pelo Git, um sistema de controle de versão. Isso geralmente inclui arquivos
temporários, arquivos de compilação e outros arquivos que não devem ser versionados.

|-- uncommitted # Este é um diretório onde você pode colocar arquivos que ainda não foram
confirmados no controle de versão. Isso pode incluir arquivos de configuração, documentação ou
resultados de validação que ainda não foram revisados ou aprovados para inclusão no
repositório principal

|-- config_variables.yml # Este arquivo YAML pode conter variáveis de configuração
específicas do ambiente ou do projeto. Por exemplo, você pode definir variáveis de ambiente ou
configurações de conexão de banco de dados aqui.

|-- data_docs # Este diretório pode conter documentação gerada automaticamente sobre os
dados, incluindo estatísticas, perfis de colunas e outras informações úteis para entender o
conjunto de dados

|-- validations # Este diretório pode conter resultados de validações anteriores, como logs ou
relatórios de validação, que podem ser úteis para rastrear a qualidade dos dados ao longo do
tempo.
...
```

A base que iremos extrair

Configurando o projeto

Agora que criamos o projeto podemos alterar o arquivo yml great_expectations na raiz do projeto, adicionando o json a seguir para configurar o autos_cleansed.csv como um datasource

Execute:

```
great_expectations datasource new
```

Escolha a opção 1 pandas

```

As of V0.16, the preferred method for adding a Datasource is using the fluent method of c
onfiguration, and not using the CLI.
You can read more about this here: https://greatexpectations.io/blog/the-fluent-way-to-co
nnect-to-data-sources-in-gx
If you would like to proceed anyway, press Y. [Y/n]: Y

What data would you like Great Expectations to connect to?
  1. Files on a filesystem (for processing with Pandas or Spark)
  2. Relational database (SQL)
: 1

What are you processing your files with?
  1. Pandas
  2. PySpark
: 1

```

Depois passe o caminho do .csv absoluto até a pasta onde estará o arquivo

```

Enter the path of the root directory where the data files are stored. If files are on loc
al disk enter a path relative to your current working directory or an absolute path.
: C:\Users\costa\workspace\ada-tech\engenharia-de-dados-cursos\analytics-engineering\aula
_5\dados

```

Execute o jupyter notebook que o great expectations abrir, que o arquivo yml vai ser automaticamente modificado

Create a new pandas Datasource

Use this notebook to configure a new pandas Datasource and add it to your project.

```
[1]: import great_expectations as gx
from great_expectations.cli.datasource import sanitize_yaml_and_save_datasource, check_if_datasource_name_exists
context = gx.get_context()
```

Customize Your Datasource Configuration

If you are new to Great Expectations Datasources, you should check out our [how-to documentation](#)

My configuration is not so simple - are there more advanced options? Glad you asked! Datasources are versatile. Please see our [How To Guides!](#)

Give your datasource a unique name:

```
[2]: datasource_name = "my_datasource"
```

For files based Datasources:

Here we are creating an example configuration. The configuration contains an **InferredAssetFilesystemDataConnector** which will add a Data Asset for each file in the base directory you provided. It also contains a **RuntimeDataConnector** which can accept filepaths. This is just an example, and you may customize this as you wish!

Also, if you would like to learn more about the **DataConnectors** used in this configuration, including other methods to organize assets, handle multi-file assets, name assets based on parts of a filename, please see our docs on [InferredAssetDataConnectors](#) and [RuntimeDataConnectors](#).

A célula do yml é a mais importante pois será a que realizará a configuração no yml principal do great expectations de um novo data source

```
[3]: example_yaml = f"""
name: {datasource_name}
class_name: Datasource
execution_engine:
  class_name: PandasExecutionEngine
data_connectors:
  default_inferred_data_connector_name:
    class_name: InferredAssetFilesystemDataConnector
    base_directory: ../dados
    default_regex:
      group_names:
        - data_asset_name
      pattern: (.*)
  default_runtime_data_connector_name:
    class_name: RuntimeDataConnector
assets:
  my_runtime_asset_name:
    batch_identifiers:
      - runtime_batch_identifier_name
"""
print(example_yaml)
```

```
name: my_datasource
class_name: Datasource
execution_engine:
  class_name: PandasExecutionEngine
data_connectors:
  default_inferred_data_connector_name:
    class_name: InferredAssetFilesystemDataConnector
    base_directory: ../dados
    default_regex:
      group_names:
        - data_asset_name
      pattern: (.*)
  default_runtime_data_connector_name:
    class_name: RuntimeDataConnector
assets:
  my_runtime_asset_name:
    batch_identifiers:
      - runtime_batch_identifier_name
```

É possível também testar esse yaml no contexto do framework

```
[4]: context.test_yaml_config(yaml_config=example_yaml)

Attempting to instantiate class from config...
  Instantiating as a Datasource, since class_name is Datasource
  Successfully instantiated Datasource

ExecutionEngine class name: PandasExecutionEngine
Data Connectors:
  default_inferred_data_connector_name : InferredAssetFilesystemDataConnector

  Available data_asset_names (3 of 4):
    .ipynb_checkpoints (1 of 1): ['.ipynb_checkpoints']
    autos.csv (1 of 1): ['autos.csv']
    autos_cleaned.csv (1 of 1): ['autos_cleaned.csv']

  Unmatched data_references (0 of 0):[]

  default_runtime_data_connector_name:RuntimeDataConnector

  default_runtime_data_connector_name : RuntimeDataConnector

  Available data_asset_names (1 of 1):
    my_runtime_asset_name (0 of 0): []

  Unmatched data_references (0 of 0):[]

[4]: <great_expectations.datasource.new_datasource.Datasource at 0x1e7c5969310>
```

E a última célula executável realiza um commit da modificação no yaml

Save Your Datasource Configuration ¶

Here we will save your Datasource in your Data Context once you are satisfied with the configuration. Note that `overwrite_existing` defaults to False, but you may change it to True if you wish to overwrite. Please note that if you wish to include comments you must add them directly to your `great_expectations.yml`.

```
[5]: sanitize_yaml_and_save_datasource(context, example_yaml, overwrite_existing=False)
context.list_datasources()

**WARNING** A Datasource named "my_datasource" already exists in this Data Context. The Datasource has *not* been saved. Please use a different name or set overwrite_existing=True if you want to overwrite!

[5]: [{'name': 'my_datasource',
      'class_name': 'Datasource',
      'module_name': 'great_expectations.datasource',
      'execution_engine': {'class_name': 'PandasExecutionEngine',
                           'module_name': 'great_expectations.execution_engine'},
      'data_connectors': {'default_inferred_data_connector_name': {'class_name': 'InferredAssetFileSystemDataConnector',
                                                                  'module_name': 'great_expectations.datasource.data_connector',
                                                                  'base_directory': '..\\dados',
                                                                  'default_regex': {'group_names': ['data_asset_name'], 'pattern': '(.*?)'},
                                                                  'default_runtime_data_connector_name': {'class_name': 'RuntimeDataConnector',
                                                                                   'module_name': 'great_expectations.datasource.data_connector',
                                                                                   'assets': {'my_runtime_asset_name': {'class_name': 'Asset',
                                                                                   'module_name': 'great_expectations.datasource.data_connector.asset',
                                                                                   'batch_identifiers': ['runtime_batch_identifier_name']}}}}}]

Now you can close this notebook and delete it!
```

O yml estará alterado conforme previsto

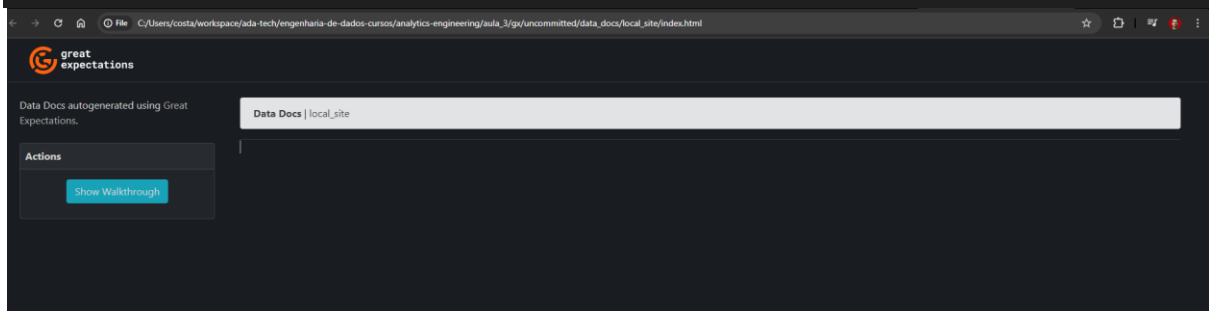
```
la_3 > gx > ! great_expectations.yml
7  # - Join our slack channel: http://greatexpectations.io/slack
8
9  # config_version refers to the syntactic version of this config file, and is used to
10 # It is auto-generated and usually does not need to be changed.
11 config_version: 3.0
12
13 # Datasources tell Great Expectations where your data lives and how to get it.
14 # Read more at https://docs.greatexpectations.io/docs/guides/connecting\_to\_your\_data\_sources
15 datasources:
16   my_datasource:
17     class_name: Datasource
18     module_name: great_expectations.datasource
19     execution_engine:
20       class_name: PandasExecutionEngine
21       module_name: great_expectations.execution_engine
22     data_connectors:
23       default_inferred_data_connector_name:
24         class_name: InferredAssetFileSystemDataConnector
25         module_name: great_expectations.datasource.data_connector
26         base_directory: ..\\dados
27         default_regex:
28           group_names:
29             - data_asset_name
30           pattern: (.*?)
31       default_runtime_data_connector_name:
32         class_name: RuntimeDataConnector
33         module_name: great_expectations.datasource.data_connector
34     assets:
35       my_runtime_asset_name:
36         class_name: Asset
37         module_name: great_expectations.datasource.data_connector.asset
38         batch_identifiers:
39           - runtime_batch_identifier_name
40
```

Após executar esses comandos o datasource estará criado, e então poderemos criar a nossa suite de testes

Buildando a documentação pela primeira vez

Agora que já definimos o arquivo csv como datasource, vamos configurar as nossas expectations, mas antes disso vamos iniciar o buildar nosso projeto, executando o comando abaixo:

```
great_expectations docs build
```

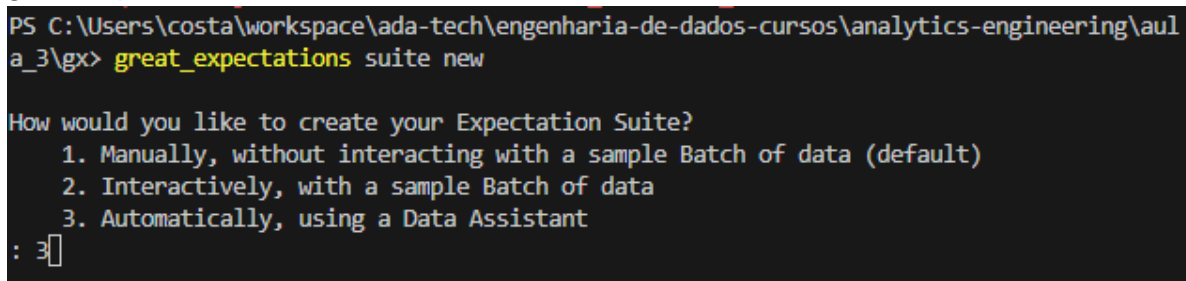


Podemos ver que não existe nenhuma suite de testes, ou execução criada.

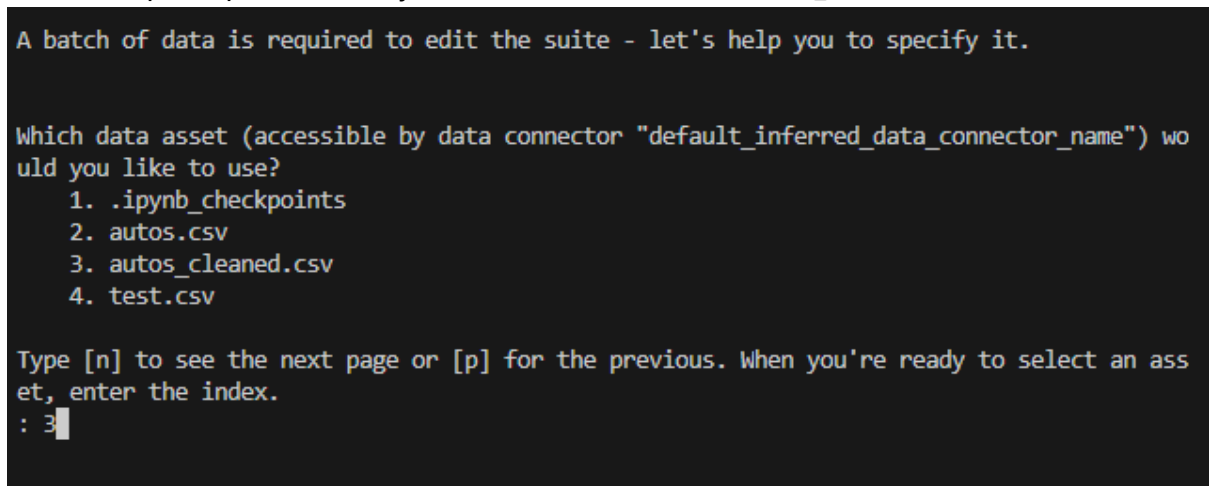
Criando as expectativas

Execute o comando

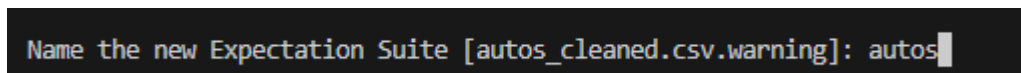
```
great_expectations suite new
```



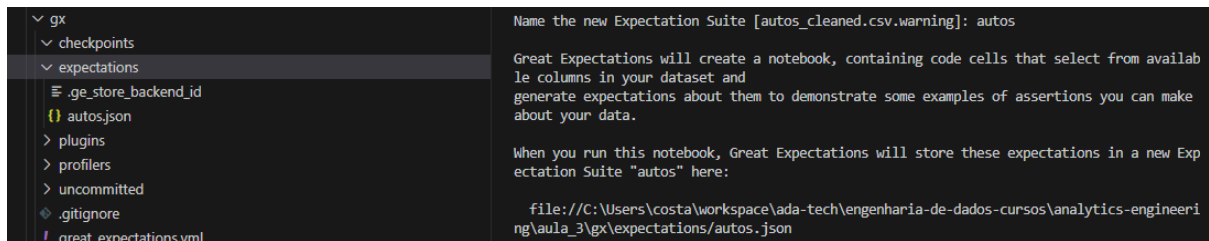
Passe o arquivo que você deseja utilizar, no nosso caso o autos_cleaned.csv



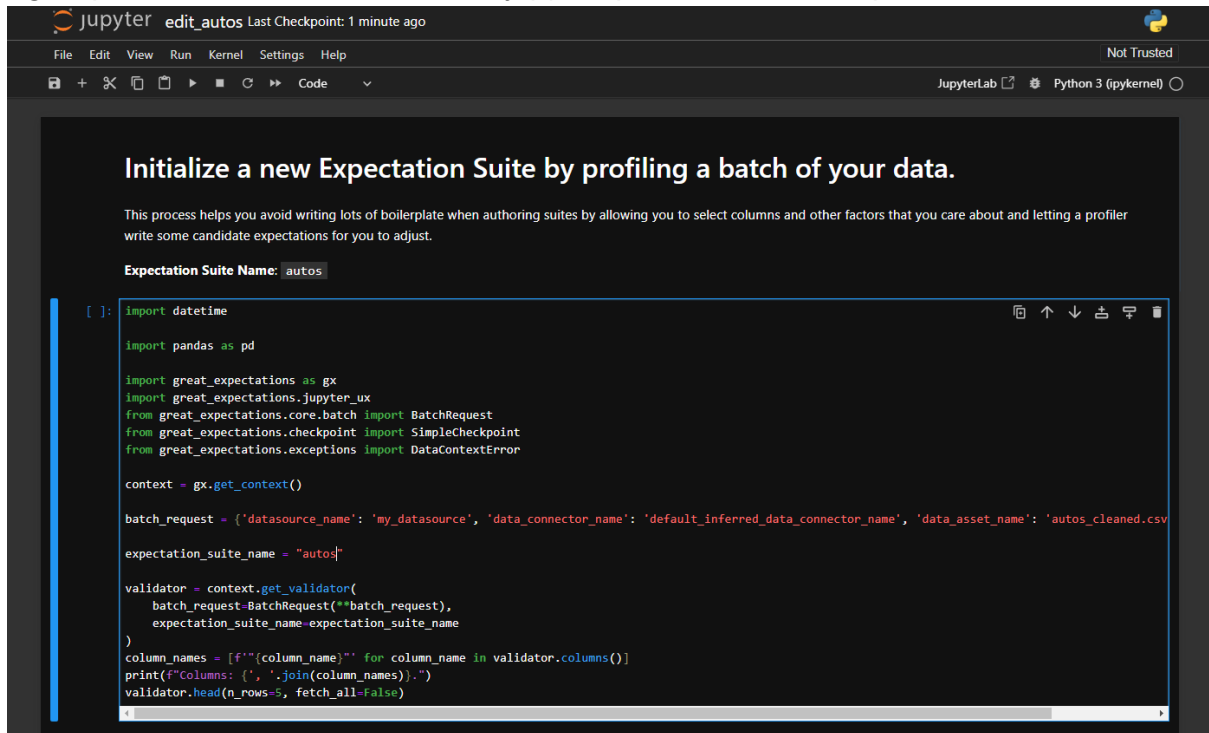
E então nomeie a suite de testes



Aperte Y para proceder e então será aberto um novo jupyter, a ideia desse é testar algumas expectations padrão de profiling, como o do y_data_profiling que vimos em aulas anteriores, além disso o arquivo autos.json também foi criado.

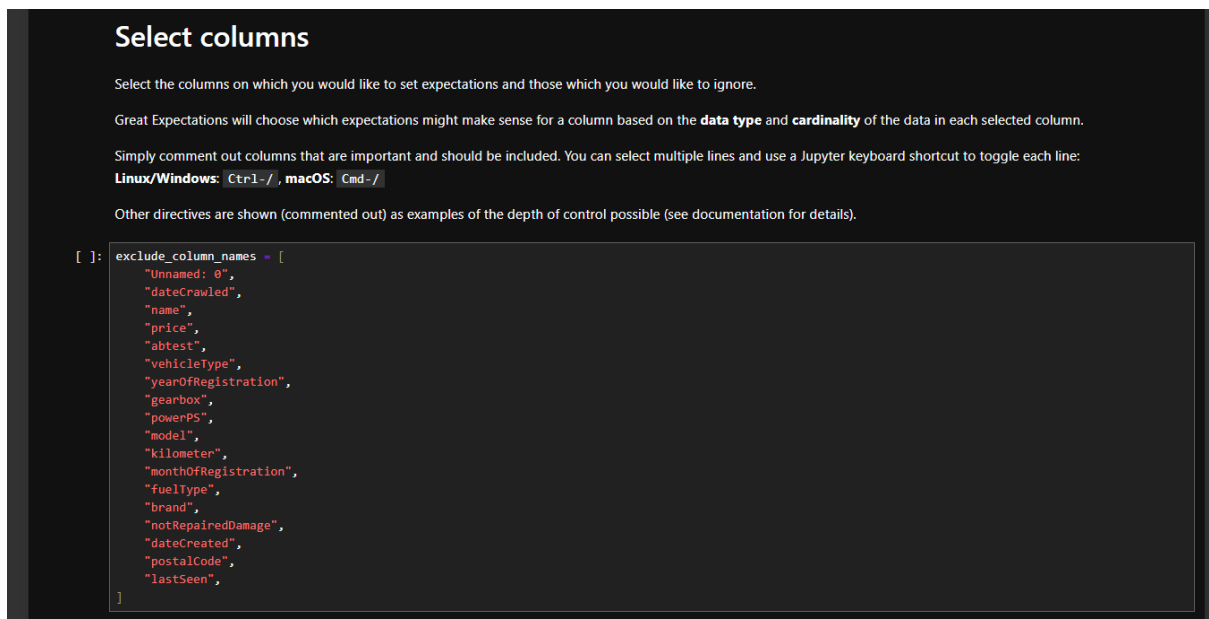


Agora podemos executar as células do jupyter que são bem auto explicativa



Essa primeira instancia um validador para criar a suite de fato

A segunda, nós podemos tirar as colunas ou adicionar, que queremos criar testes, eu vou tirar todas



Select columns

Select the columns on which you would like to set expectations and those which you would like to ignore.

Great Expectations will choose which expectations might make sense for a column based on the **data type** and **cardinality** of the data in each selected column.

Simply comment out columns that are important and should be included. You can select multiple lines and use a Jupyter keyboard shortcut to toggle each line:

Linux/Windows: `Ctrl-/,` **macOS:** `Cmd-/,`

Other directives are shown (commented out) as examples of the depth of control possible (see documentation for details).

```
[ ]: exclude_column_names = []
```

Cria a suite com as colunas que queremos, entao ele vai gerar uma serie de expectations para as colunas da tabela

Run the OnboardingDataAssistant

The suites generated here are **not meant to be production suites** -- they are a **starting point to build upon**.

To get to a production-grade suite, you will definitely want to [edit this suite](#) after this initial step gets you started on the path towards what you want.

This is highly configurable depending on your goals. You can ignore columns, specify cardinality of categorical columns, configure semantic types for columns, even adjust thresholds and/or different estimator parameters, etc. You can find more information about OnboardingDataAssistant and other DataAssistant components (please see documentation for the complete set of DataAssistant controls) [how to choose and control the behavior of the DataAssistant tailored to your goals](#).

Performance considerations:

- Latency: We optimized for an explicit "question/answer" design, which means we issue **lots** of queries. Connection latency will impact performance.
- Data Volume: Small samples of data will often give you a great starting point for understanding the dataset. Consider configuring a sampled asset and profiling a small number of batches.

```
[ ]: result = context.assistants.onboarding.run(
    batch_request=batch_request,
    exclude_column_names=exclude_column_names,
)
validator.expectation_suite = result.get_expectation_suite(
    expectation_suite_name=expectation_suite_name
)
```

Esse passo pode demorar um pouco

```
[*]: result = context.assistants.onboarding.run(
    batch_request=batch_request,
    exclude_column_names=exclude_column_names,
)
validator.expectation_suite = result.get_expectation_suite(
    expectation_suite_name=expectation_suite_name
)
```

Generating Expectations: 62%	<div></div>	5/8 [00:03<00:02, 1.26it/s]
Calculating Metrics: 100%	<div></div>	1/1 [00:00<00:00, 200.00it/s]
Calculating Metrics: 100%	<div></div>	2/2 [00:00<00:00, 333.41it/s]
Calculating Metrics: 100%	<div></div>	2/2 [00:00<00:00, 399.99it/s]
Calculating Metrics: 100%	<div></div>	2/2 [00:00<00:00, 400.01it/s]

Agora vamos salvar o resultado da nossa suite, ele inclusive cria um checkpoint

Save & review your new Expectation Suite

Let's save the draft expectation suite as a JSON file in the `great_expectations/expectations` directory of your project and rebuild the Data Docs site to make it easy to review your new suite.

```
[ ]: print(validator.get_expectation_suite(discard_failed_expectations=False))
validator.save_expectation_suite(discard_failed_expectations=False)

checkpoint_config = {
    "class_name": "SimpleCheckpoint",
    "validations": [
        {
            "batch_request": batch_request,
            "expectation_suite_name": expectation_suite_name
        }
    ]
}
checkpoint = SimpleCheckpoint(
    f'{validator.active_batch_definition.data_asset_name}_{expectation_suite_name}',
    context,
    **checkpoint_config
)
checkpoint_result = checkpoint.run()

context.build_data_docs()

validation_result_identifier = checkpoint_result.list_validation_result_identifiers()[0]
context.open_data_docs(resource_identifier=validation_result_identifier)
```

Podemos ver que foram executadas mais de 100 expectations

```
2024-04-18T18:38:42-0300 - INFO - 110 expectation(s) included in expectation_suite.
{
  "expectation_suite_name": "autos",
  "ge_cloud_id": null,
  "expectations": [
    {
      "expectation_type": "expect_table_row_count_to_be_between",
      "kwargs": {
        "min_value": 303950,
        "max_value": 303950
      },
      "meta": {
        "profiler_details": {
          "metric_configuration": {
            "metric_name": "table.row_count",
            "domain_kwargs": {},
            "metric_value_kwargs": null
          }
        }
      }
    }
  ]
}
```

É possível alterar a suite com o comando

Next steps

After you review this initial Expectation Suite in Data Docs you should edit this suite to make finer grained adjustments to the expectations. This can be done by running `great_expectations suite edit autos`.

Mas também é possível alterar via yml

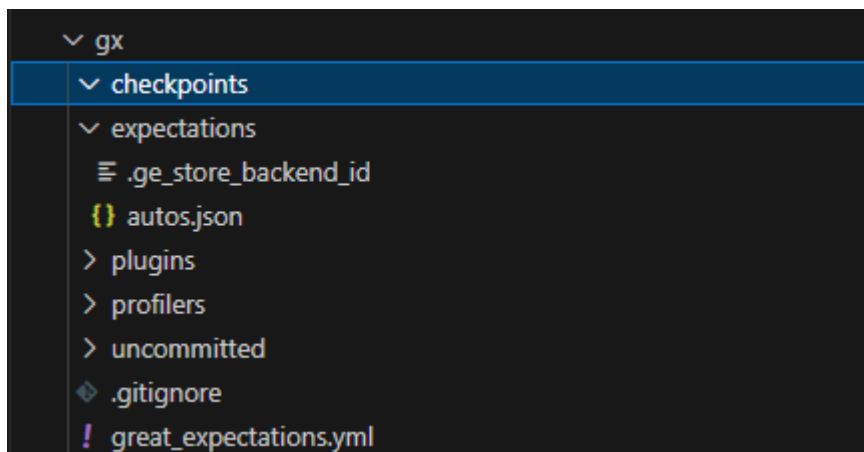
Vamos executar o comando

`great_expectations docs build`

Para avaliarmos a suite que foi executada

Status	Run Time	Run Name	Asset Name	Batch ID	Expectation Suite
✓	2024-04-18 18:38:42 -03	._none_		1b490643eea781090fc37ba896879114	autos

Podemos ver, que o checkpoint foi executado, mas nesse caso não foi criado um yml



Podemos verificar que o gx conseguiu criar 110 expectativas automáticas que tiveram 100% de sucesso para todas as colunas da tabela, isso mostra o poder da ferramenta que permite uma cobertura de testes enorme com pouco esforço de configuração e de forma simples

Expectation Validation Result
Evaluates whether a batch of data matches expectations.

Overview
Expectation Suite: `autos`
Data asset: None
Status: ✔ Succeeded

Statistics

Statistic	Value
Evaluated Expectations	110
Successful Expectations	110
Unsuccessful Expectations	0
Success Percent	100%

Show more info...

Table-Level Expectations

Status	Expectation	Observed Value
✔	Must have greater than or equal to <code>303950</code> and less than or equal to <code>303950</code> rows.	303950
✔	Must have at least these columns (in any order): <code>dateCreated</code> , <code>yearOfRegistration</code> , <code>powerPS</code> , <code>monthOfRegistration</code> , <code>notRepairedDamage</code> , <code>lastSeen</code> , <code>postalCode</code> , <code>kilometer</code> , <code>Unnamed: 0</code> , <code>gearbox</code> , <code>dateCrawled</code> , <code>abtest</code> , <code>vehicleType</code> , <code>brand</code> , <code>name</code> , <code>price</code> , <code>fuelType</code> , <code>model</code>	[<code>Unnamed: 0</code> , <code>dateCrawled</code> , <code>name</code> , <code>price</code> , <code>abtest</code> , <code>vehicleType</code> , <code>yearOfRegistration</code> , <code>gearbox</code> , <code>powerPS</code> , <code>model</code> , <code>kilometer</code> , <code>monthOfRegistration</code> , <code>fuelType</code> , <code>brand</code> , <code>notRepairedDamage</code> , <code>dateCreated</code> , <code>postalCode</code> , <code>lastSeen</code>]

abtest

Agora vamos criar um checkpoint para deixar parametriza e reutilizável, o que utilizamos foi somente em memória via jupyter

```
great_expectations checkpoint new checkpoint_autos
```

Novamente vamos para uma interface jupyter

jupyter edit_checkpoint_checkpoint_autos Last Checkpoint: 13 seconds ago

File Edit View Run Kernel Settings Help Not Trusted

JupyterLab Python 3 (ipykernel)

Create Your Checkpoint

Use this notebook to configure a new Checkpoint and add it to your project:

Checkpoint Name: `checkpoint_autos`

```
[ ]: from ruamel.yaml import YAML
import great_expectations as gx
from pprint import pprint

yaml = YAML()
context = gx.get_context()
```

Create a Checkpoint Configuration

If you are new to Great Expectations or the Checkpoint feature, you should start with SimpleCheckpoint because it includes default configurations like a default list of post validation actions.

In the cell below we have created a sample Checkpoint configuration using **your configuration** and **SimpleCheckpoint** to run a single validation of a single Expectation Suite against a single Batch of data.

To keep it simple, we are just choosing the first available instance of each of the following items you have configured in your Data Context:

- DataSource
- DataConnector
- DataAsset
- Partition
- Expectation Suite

Of course this is purely an example, you may edit this to your heart's content.

My configuration is not so simple - are there more advanced options?

Glad you asked! Checkpoints are very versatile. For example, you can validate many Batches in a single Checkpoint, validate Batches against different Expectation Suites or

E vamos executar novamente as células, que gerarão nosso checkpoint final

Glad you asked! Checkpoints are very versatile. For example, you can validate many Batches in a single Checkpoint, validate Batches against different Expectation Suites or against many Expectation Suites, control the specific post-validation actions based on Expectation Suite / Batch / results of validation among other features. Check out our documentation on Checkpoints for more details and for instructions on how to implement other more advanced features including using the **Checkpoint** class:

- https://docs.greatexpectations.io/docs/reference/checkpoints_and_actions
- https://docs.greatexpectations.io/docs/guides/validation/checkpoints/how_to_create_a_new_checkpoint
- https://docs.greatexpectations.io/docs/guides/validation/checkpoints/how_to_configure_a_new_checkpoint_using_test_yaml_config

```
[2]: my_checkpoint_name = "checkpoint_autos" # This was populated from your CLI command.

yaml_config = f"""
name: {my_checkpoint_name}
config_version: 1.0
class_name: SimpleCheckpoint
run_name_template: "%Y%m%d-%H%M%S-my-run-name-template"
validations:
  - batch_request:
      datasource_name: my_datasource
      data_connector_name: default_inferred_data_connector_name
      data_asset_name: autos_cleaned.csv
      data_connector_query:
        index: -1
      expectation_suite_name: autos
"""

print(yaml_config)

name: checkpoint_autos
config_version: 1.0
class_name: SimpleCheckpoint
run_name_template: "%Y%m%d-%H%M%S-my-run-name-template"
validations:
  - batch_request:
      datasource_name: my_datasource
      data_connector_name: default_inferred_data_connector_name
      data_asset_name: autos_cleaned.csv
      data_connector_query:
        index: -1
      expectation_suite_name: autos
```

Yaml gerado

Customize Your Configuration

The following cells show examples for listing your current configuration. You can replace values in the sample configuration with these values to customize your Checkpoint.

```
[ ]: # Run this cell to print out the names of your Datasources, Data Connectors and Data Assets
      pprint(context.get_available_data_asset_names())

[ ]: context.list_expectation_suite_names()
```

Test Your Checkpoint Configuration

Here we will test your Checkpoint configuration to make sure it is valid.

This `test_yaml_config()` function is meant to enable fast dev loops. If your configuration is correct, this cell will show a message that you successfully instantiated a Checkpoint. You can continually edit your Checkpoint config yaml and re-run the cell to check until the new config is valid.

If you instead wish to use python instead of yaml to configure your Checkpoint, you can use `context.add_checkpoint()` and specify all the required parameters.

```
[ ]: my_checkpoint = context.test_yaml_config(yaml_config-yaml_config)
```

Review Your Checkpoint

You can run the following cell to print out the full yaml configuration. For example, if you used **SimpleCheckpoint** this will show you the default action list.

```
[ ]: print(my_checkpoint.get_config(mode="yaml"))
```

Add Your Checkpoint

Run the following cell to save this Checkpoint to your Checkpoint Store.

```
[ ]: context.add_checkpoint(**yaml.load(yaml_config))
```

É possível verificar como ficara o checkpoint final

Review Your Checkpoint

You can run the following cell to print out the full yaml configuration. For example, if you used **SimpleCheckpoint** this will show you the default action list.

```
[6]: print(my_checkpoint.get_config(mode="yaml"))

name: checkpoint_autos
config_version: 1.0
template_name:
module_name: great_expectations.checkpoint
class_name: SimpleCheckpoint
run_name_template: '%Y%m%d-%H%M%S-my-run-name-template'
expectation_suite_name:
batch_request: {}
action_list:
- name: store_validation_result
  action:
    class_name: StoreValidationResultAction
- name: store_evaluation_params
  action:
    class_name: StoreEvaluationParametersAction
- name: update_data_docs
  action:
    class_name: UpdateDataDocsAction
evaluation_parameters: {}
runtime_configuration: {}
validations:
- batch_request:
    datasource_name: my_datasource
    data_connector_name: default_inferred_data_connector_name
    data_asset_name: autos_cleaned.csv
    data_connector_query:
      index: -1
    expectation_suite_name: autos
  profilers: []
  ge_cloud_id:
  expectation_suite_ge_cloud_id:
```

E o último comando, realiza o add checkpoint, e materializa o arquivo yaml

```
Run the following cell to save this Checkpoint to your Checkpoint Store.

[7]: context.add_checkpoint(**yaml.load(yaml_config))

[7]: {
  "action_list": [
    {
      "name": "store_validation_result",
      "action": {
        "class_name": "StoreValidationResultAction"
      }
    },
    {
      "name": "store_evaluation_params",
      "action": {
        "class_name": "StoreEvaluationParametersAction"
      }
    },
    {
      "name": "update_data_docs",
      "action": {
        "class_name": "UpdateDataDocsAction"
      }
    }
  ],
  "batch_request": {},
  "class_name": "SimpleCheckpoint",
  "config_version": 1.0,
  "evaluation_parameters": {},
  "module_name": "great_expectations.checkpoint",
  "name": "checkpoint_autos",
  "profilers": [],
  "run_name_template": "%Y%m%d-%H%M%S-my-run-name-template",
  "runtime_configuration": {},
  "validations": [
    {
      "expectation_suite_name": "autos",
      "batch_request": {
        "datasource_name": "my_datasource",
        "data_connector_name": "default_inferred_data_connector_name",
        "data_asset_name": "autos_cleaned.csv",
        "data_connector_query": {
          "index": -1
        }
      }
    }
  ]
}
```

Como podemos ver, o yml está criado corretamente na pasta checkpoints do projeto

```

1  name: checkpoint_autos
2  config_version: 1.0
3  template_name:
4  module_name: great_expectations.checkpoint
5  class_name: SimpleCheckpoint
6  run_name_template: '%Y%m%d-%H%M%S-my-run-name-template'
7  expectation_suite_name:
8  batch_request: {}
9  action_list:
10     - name: store_validation_result
11       action:
12         class_name: StoreValidationResultAction
13     - name: store_evaluation_params
14       action:
15         class_name: StoreEvaluationParametersAction
16     - name: update_data_docs
17       action:
18         class_name: UpdateDataDocsAction
19  evaluation_parameters: {}
20  runtime_configuration: {}
21  validations:
22     - batch_request:
23       datasource_name: my_datasource
24       data_connector_name: default_inferred_data_connector_name
25       data_asset_name: autos_cleaned.csv
26       data_connector_query:
27         index: -1
28       expectation_suite_name: autos
29  profilers: []
30  ge_cloud_id:
31  expectation_suite_ge_cloud_id:
32

```

Gerando o relatório uma última vez

Ao gerar o data docs novamente, podemos verificar que uma nova execução foi pontuada pelo great expectations, e agora conseguimos compartilhar esse arquivo yaml de checkpoint e trabalhar a partir dele!

Recapitulação

Na aula de hoje, exploramos o Great Expectations em um cenário mais próximo do ambiente produtivo, utilizando uma estrutura de arquivos para organizar o projeto. Começamos instalando o Great Expectations e inicializando um novo projeto. Em seguida, configuramos o datasource para um arquivo CSV específico e criamos expectativas sobre os dados.

Ao construir as expectativas, utilizamos uma interface Jupyter para testar algumas expectativas padrão de profiling e gerar um conjunto de expectativas para as colunas da tabela. Essas expectativas foram automaticamente executadas e verificadas, mostrando o poder da ferramenta para uma cobertura de testes extensa com pouco esforço de configuração.

Após criar as expectativas, construímos a documentação do projeto, permitindo uma visualização clara das execuções e resultados das validações. Geramos um checkpoint para tornar as execuções reutilizáveis e parametrizadas, o que facilita a validação contínua dos dados.

Em resumo, aprendemos a instalar o Great Expectations, inicializar um projeto, configurar o datasource, criar expectativas, construir a documentação e criar checkpoints para validações contínuas dos dados. Essa abordagem oferece uma maneira poderosa e simples de garantir a qualidade dos dados em um ambiente de análise de dados, facilitando a manutenção da confiabilidade dos dados ao longo do tempo.

Nos vemos na próxima aula!