Machine Learning Operations (MLOps)

Version Control and API Creation

Zeham Management Technologies BootCamp

by SDAIA

September the 30th, 2024

Objectives

By the end of this module, trainees will have a comprehensive understanding of:

Install and build your own repository.

Learn Imputers and Encoders

Make preprocessing Pipelines.

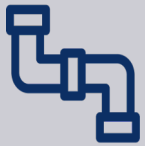
Train and validate the model.

Evaluate and save your model.

Build an API and connect it with your Machine Learning code.

Agenda

GitHub with MLOps



Data Pipelines for MLOps

Model Training

Model Validation and Evaluation

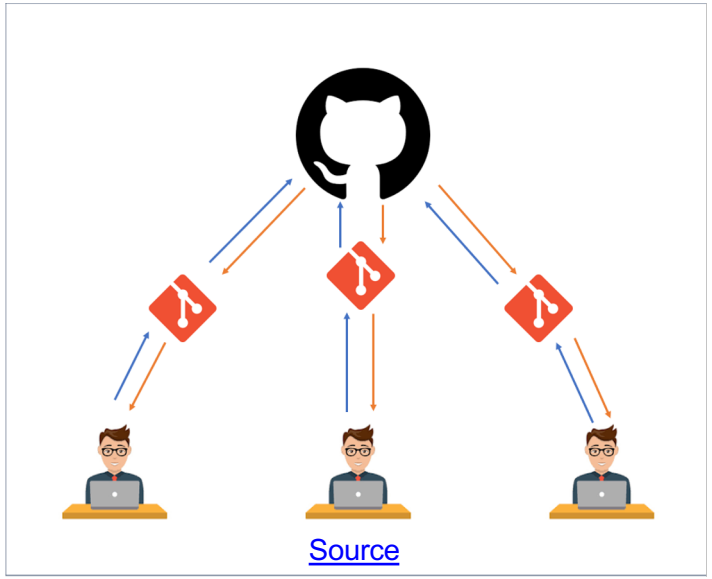
Build API for MLOps

References

GitHub with MLOps

GitHub with MLOps

GitHub is a web-based platform that uses Git, a version control system, to help developers manage code and collaborate on projects. It provides a central place to store and share code, track changes, and manage various versions of projects.

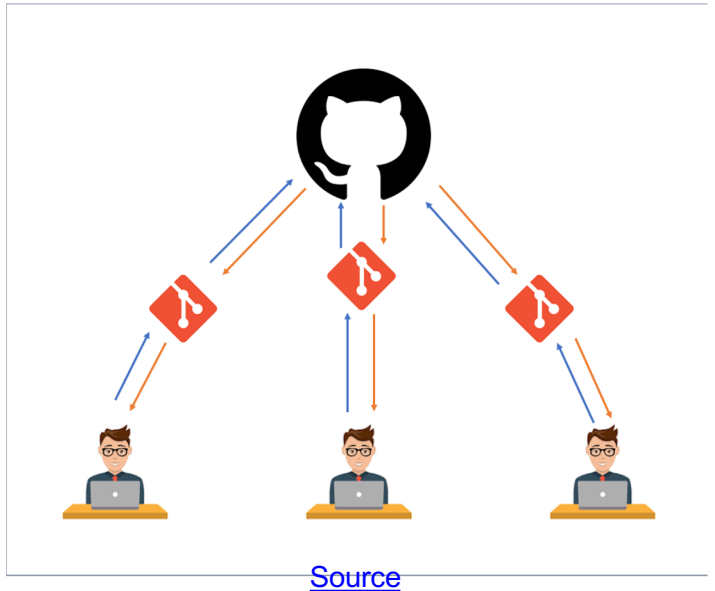


Source

GitHub with MLOps (Key Concepts)

- **Repository (Repo):** A repository is a central place where all the project files, including code, documentation, and other resources, are stored. It tracks all changes made to the files.

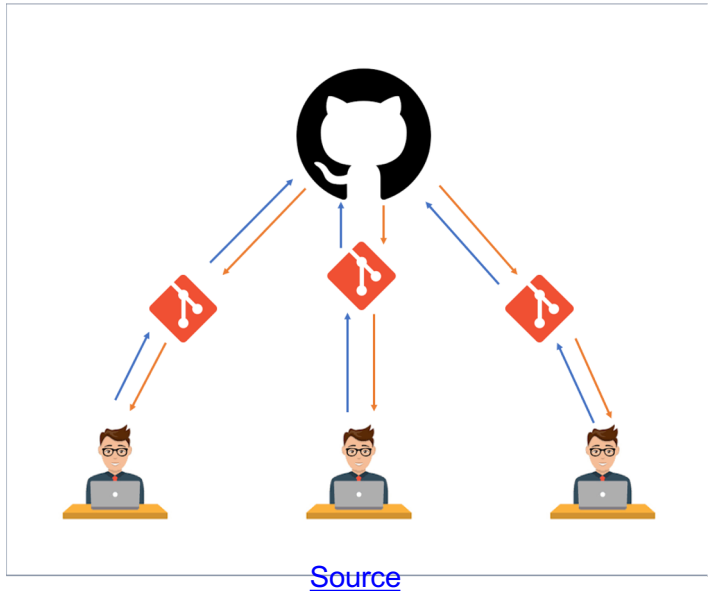
- **Branch:** A branch is a parallel version of a repository. It allows you to work on different features or fixes separately from the main (or master) branch.



Source

GitHub with MLOps (Key Concepts)

- **Commit:** A commit is a record of changes made to the repository. It acts as a checkpoint, allowing you to revert to a previous state if needed.
- **Pull Request (PR):** A pull request is a request to merge changes from one branch to another. It facilitates code review and discussion before integrating the changes.

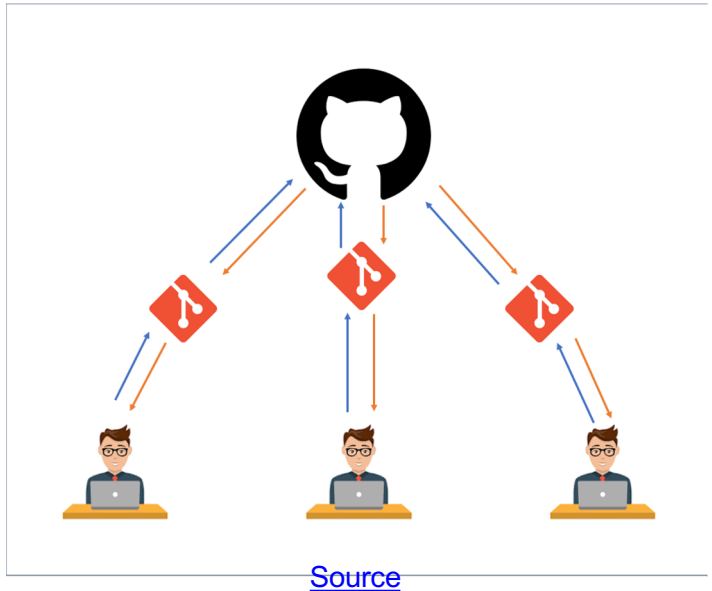


Source

GitHub with MLOps (Key Concepts)

- **Fork:** A fork is a copy of a repository that you can use to make changes without affecting the original project. It's often used to propose changes or contribute to a project you don't have write access to.

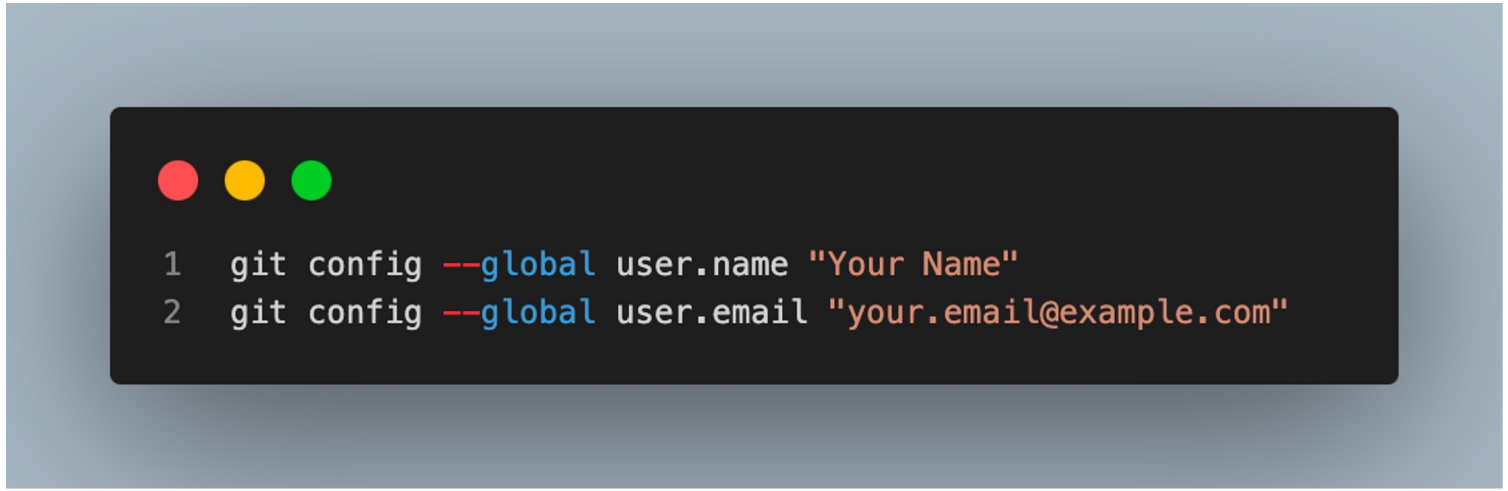
- **Clone:** Cloning is the process of creating a local copy of a repository on your machine.



Source

GitHub with MLOps (Creating Project)

We will start off by configuring Git, open your visual studio code and put these two commands in the terminal:

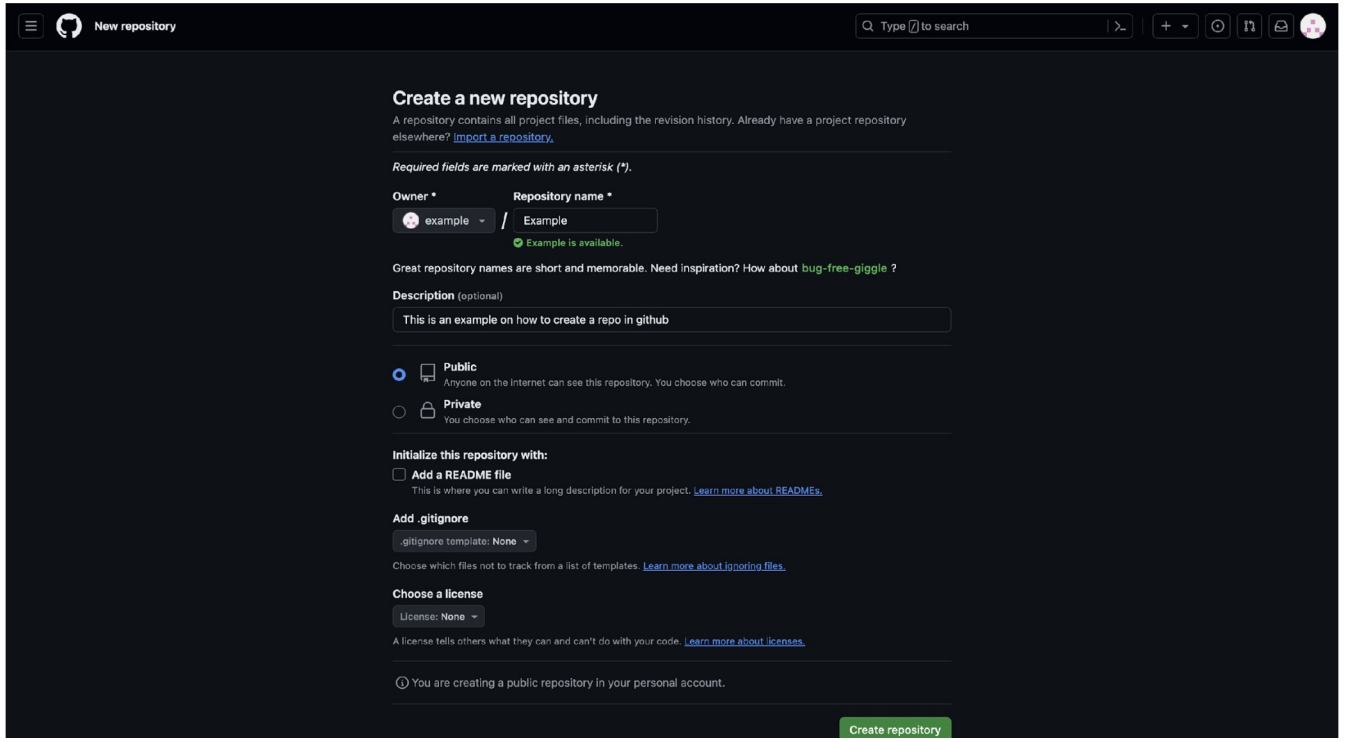


GitHub with MLOps (Creating Project)

We need to make a directory for the project either via VS Code and opening the directory, or by using the following command in the terminal:

GitHub with MLOps (Creating Project)

For the next step you will need to go to GitHub official website, log in to your account and then create the repository to clone it.



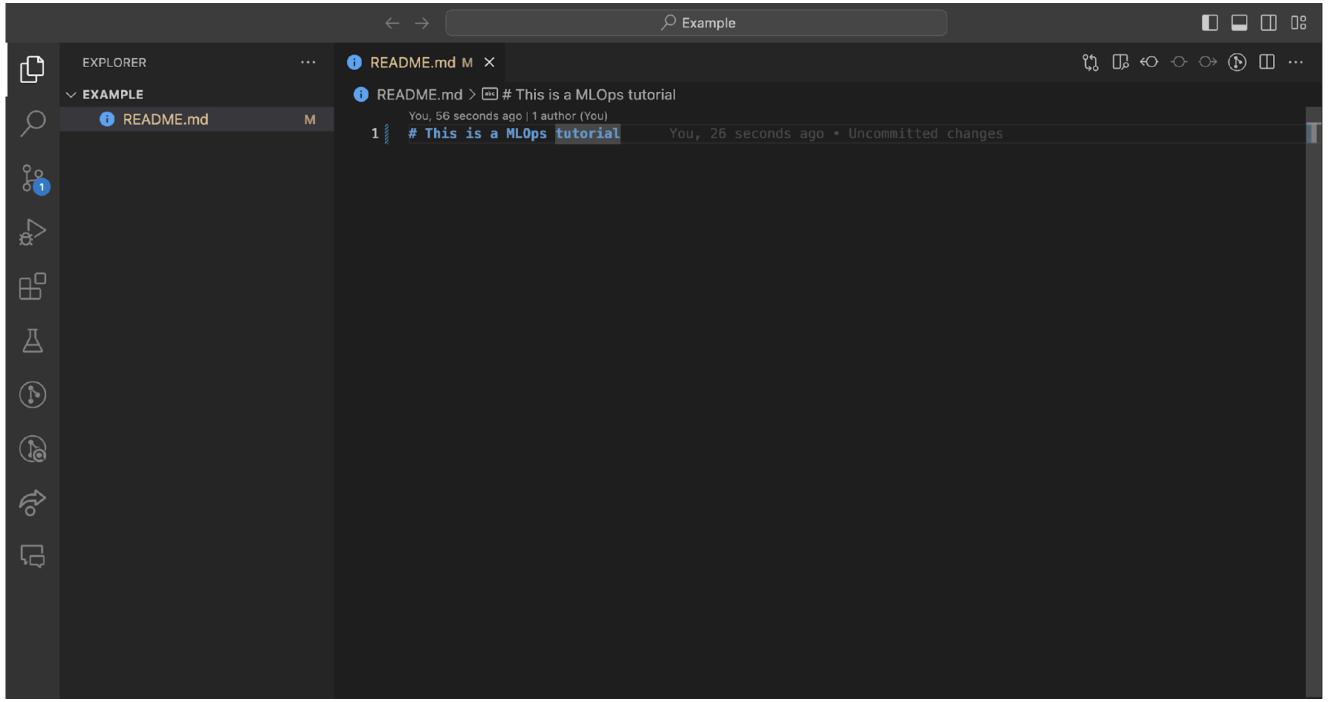
https://github.com/new

GitHub with MLOps (Creating Project)

For this step you will need to open VS Code terminal and write this command to initiate the local git repo ‘.git’:

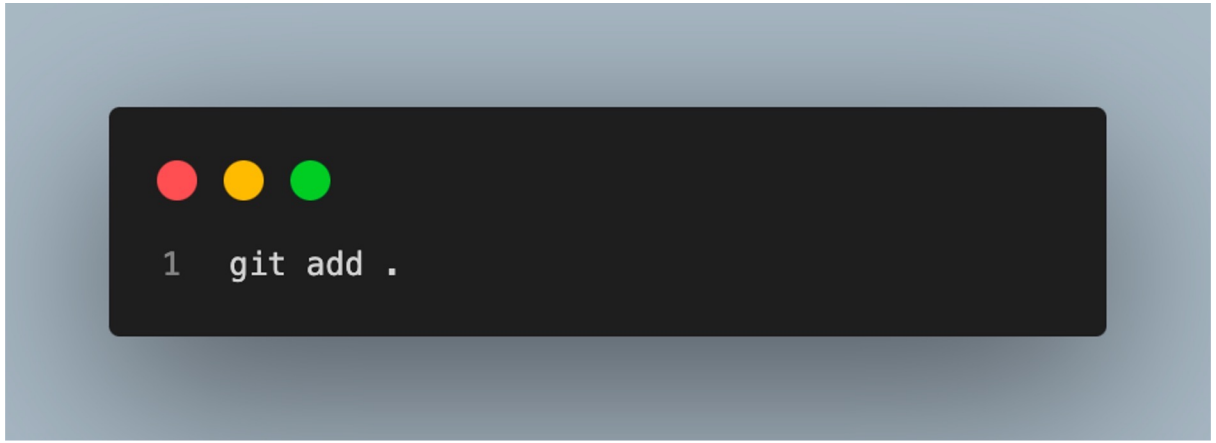
GitHub with MLOps (Creating Project)

After creating the local repo, go ahead and add a README.md with your project's files at the project's local folder.
To push to GitHub the file must not be empty.



GitHub with MLOps (Creating Project)

After filling the project's directory with your project's files use this command in the terminal to include all the current files:



GitHub with MLOps (Creating Project)

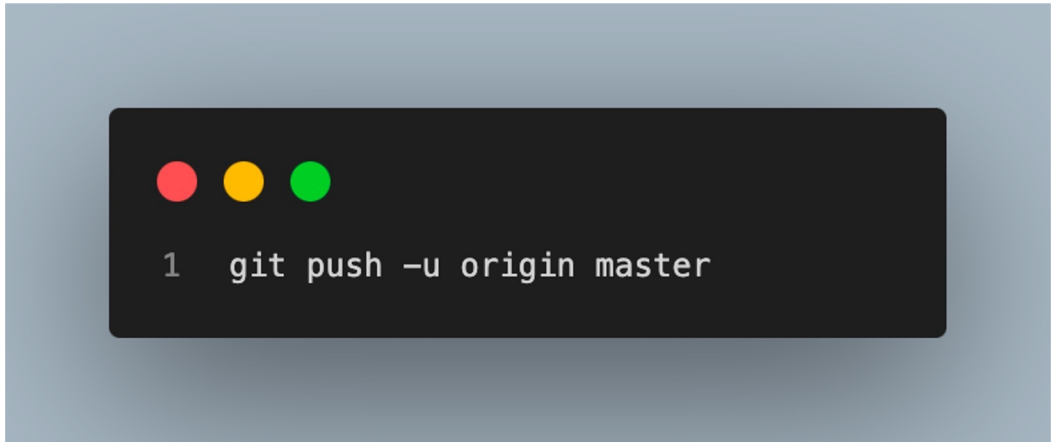
Next step is to commit your first commit using the git commit along with the commit message using (-m “message”)

GitHub with MLOps (Creating Project)

The next step is to establish a connection between your local Git repository and the GitHub repository you created. This connection allows you to push your local changes to the remote GitHub repository.

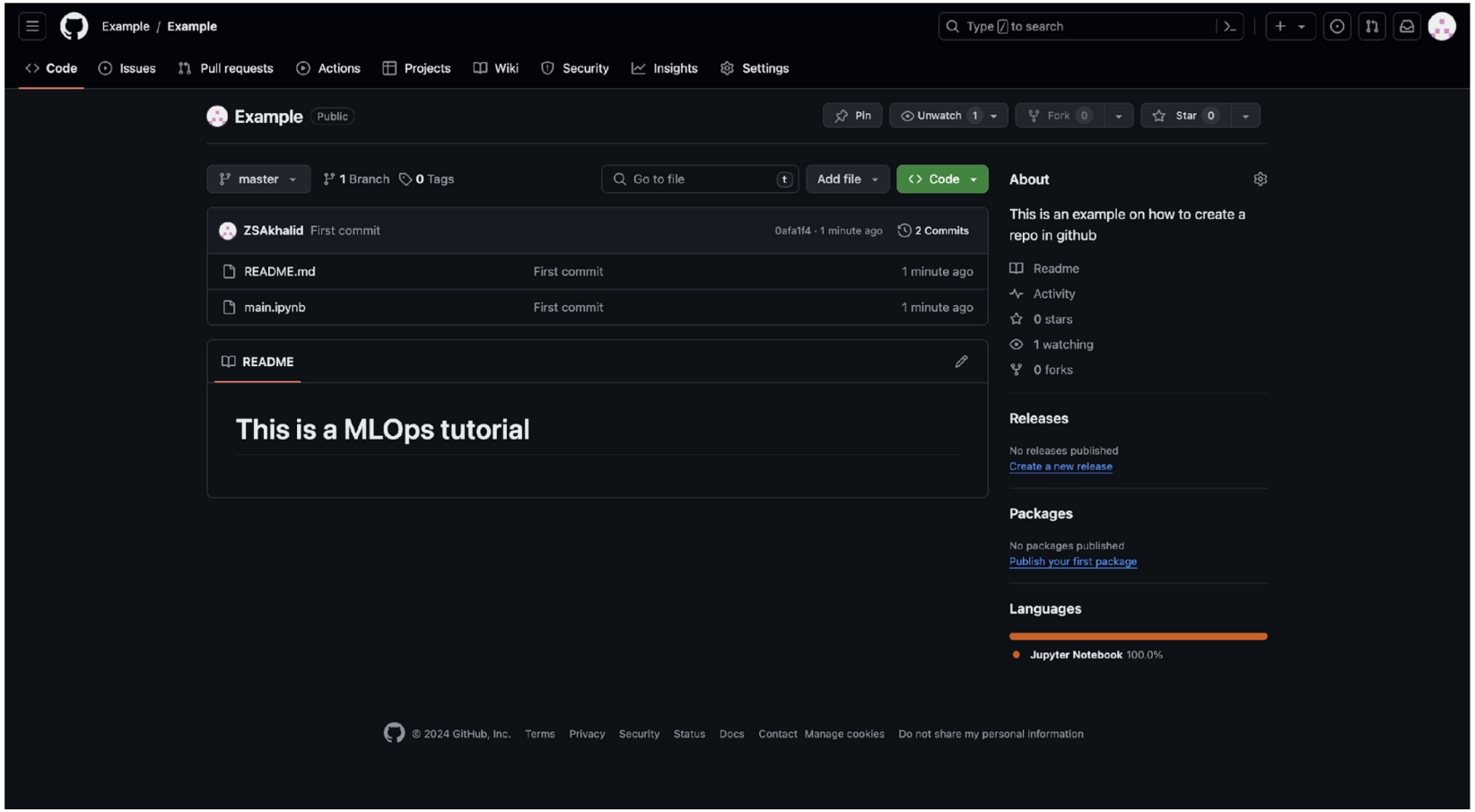
GitHub with MLOps (Creating Project)

The next step is to push your local commits to the remote GitHub repository. When you run the command, you are telling Git to push the commits from your local master branch to the master branch of the remote repository (named "origin"). Here is the command:



GitHub with MLOps (Creating Project)

Congratulations on making your first commit! 🎉



GitHub with MLOps (Creating Project)

Now if you made any changes use these commands in order:

1. Add Changes to Staging Area: Use git add to stage the changes you want to commit:
   - *git add .*
2. Commit Changes: Commit the staged changes with a commit message:
   - *git commit -m "Your commit message”*
3. Push Changes to Remote Repository:
   - *git push*

If you're pushing to the same branch on the remote repository, you can simply use git push without specifying the remote or branch name after the initial setup. If you're pushing to a different branch or remote, you would specify those in the command.

GitHub with MLOps (GitHub&Git additional commands)

Add Files to Staging Area:

GitHub with MLOps (GitHub&Git additional commands)

Push Changes to Remote Repository:

GitHub with MLOps (GitHub&Git additional commands)

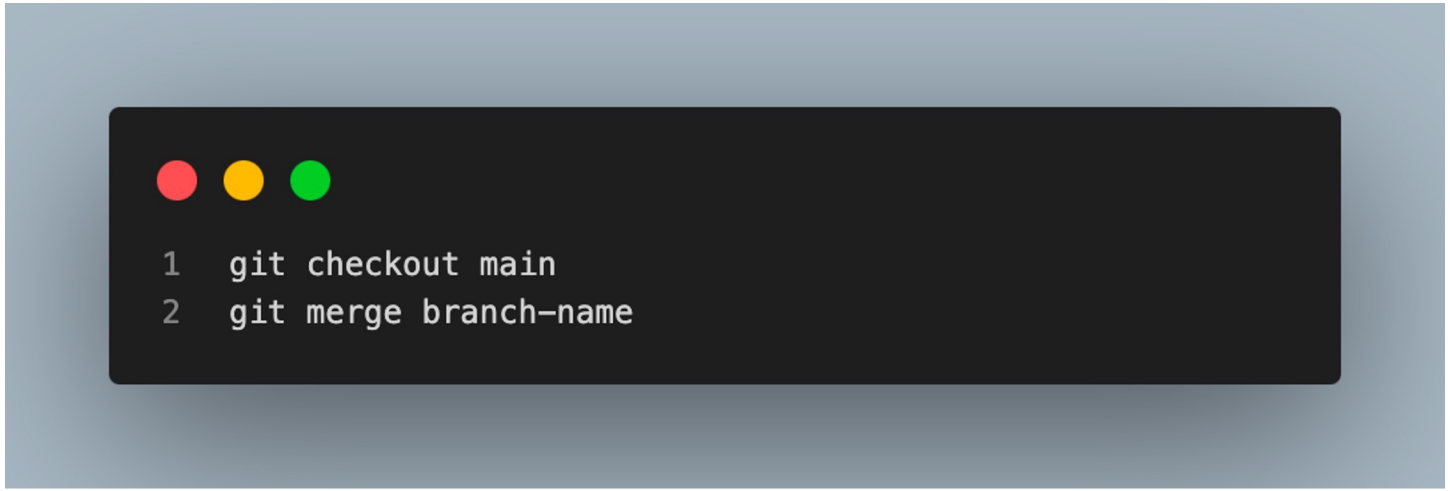
Create a New Branch:

GitHub with MLOps (GitHub&Git additional commands)

Switch to an Existing Branch:

GitHub with MLOps (GitHub&Git additional commands)

Merge a Branch:



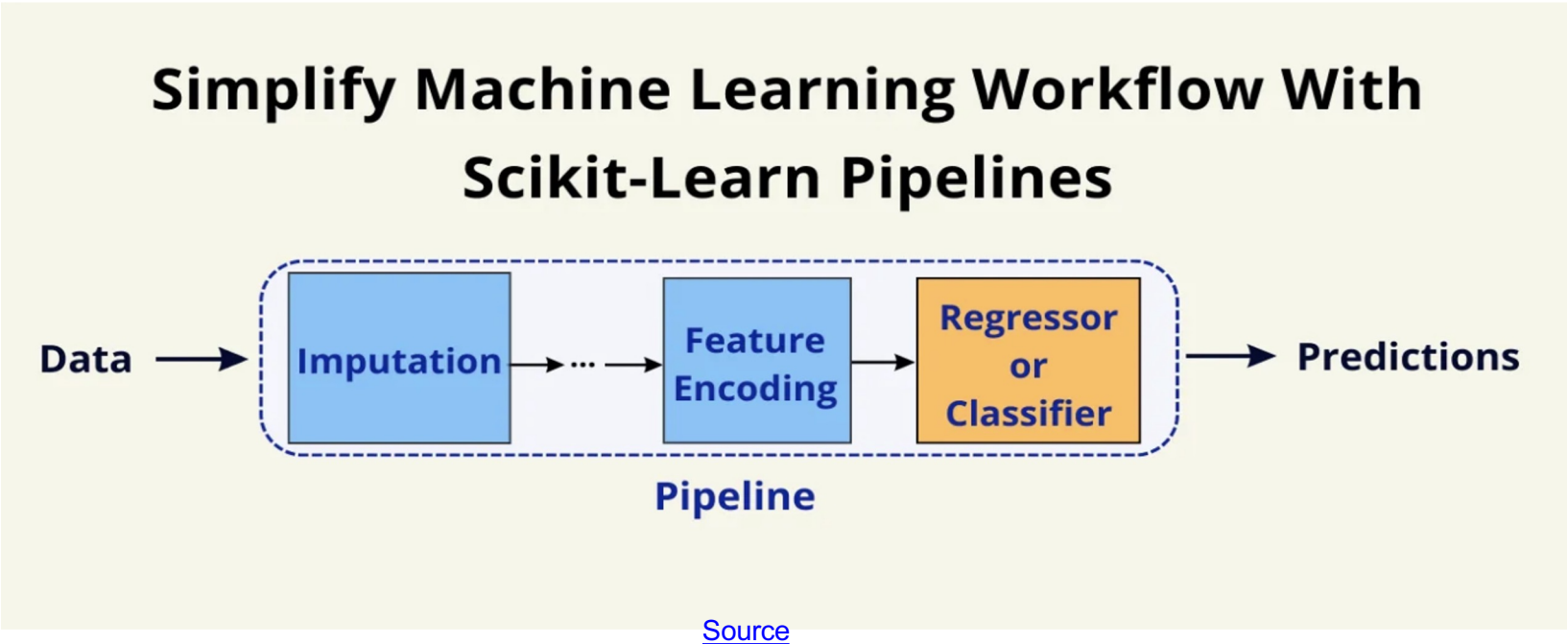
GitHub with MLOps (GitHub&Git additional commands)

Delete a Branch:

Data Pipeline for MLOps

What is a Pipeline ?

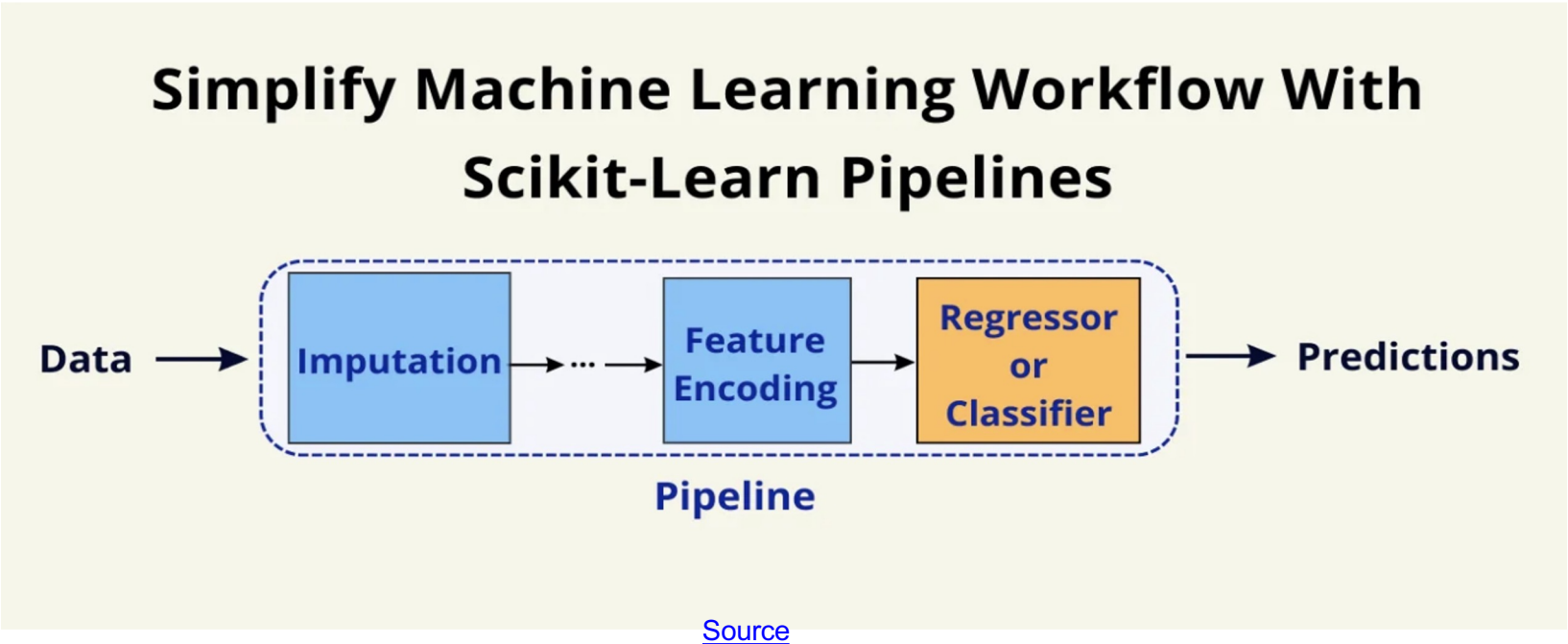
- A pipeline is a sequence of data processing steps where each step transforms the data and passes it to the next step.
- The final step typically involves fitting a machine learning model.



Source

What is a Pipeline ?

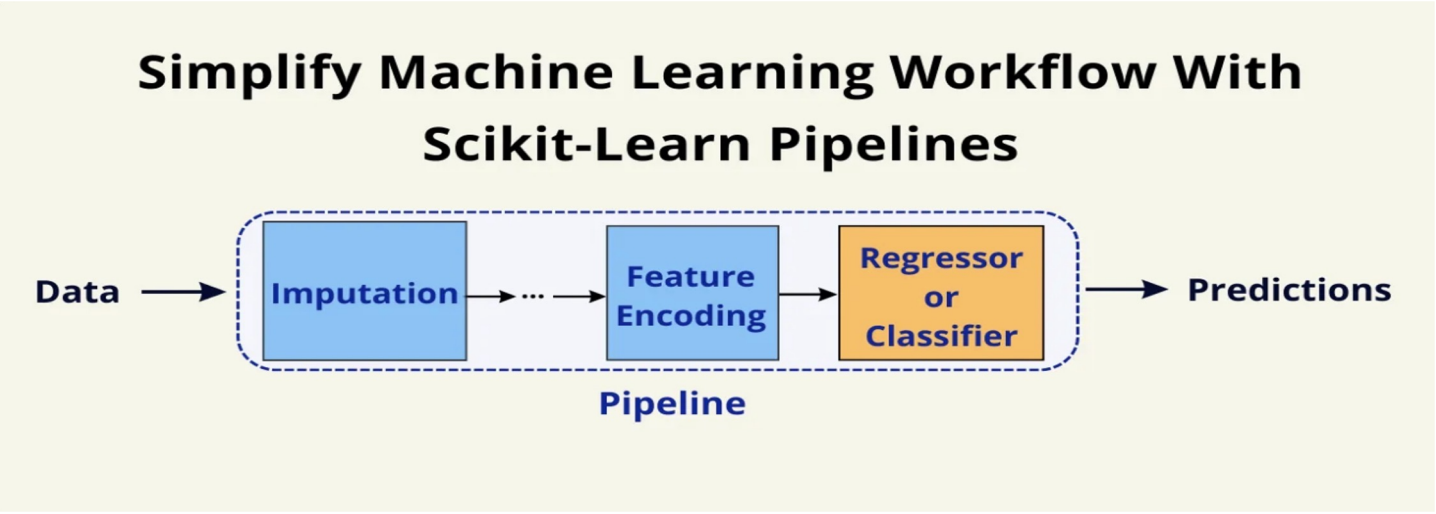
- Pipelines help streamline the machine learning workflow by combining preprocessing and modeling steps into a single object. This makes the process more manageable, especially when performing cross-validation or hyperparameter tuning.



Source

Imputer

- An imputer is a tool used to fill in missing data in a dataset. In real-world data, it's common to have some missing values.
- These missing values can create problems for machine learning algorithms, which typically require complete data to function properly.

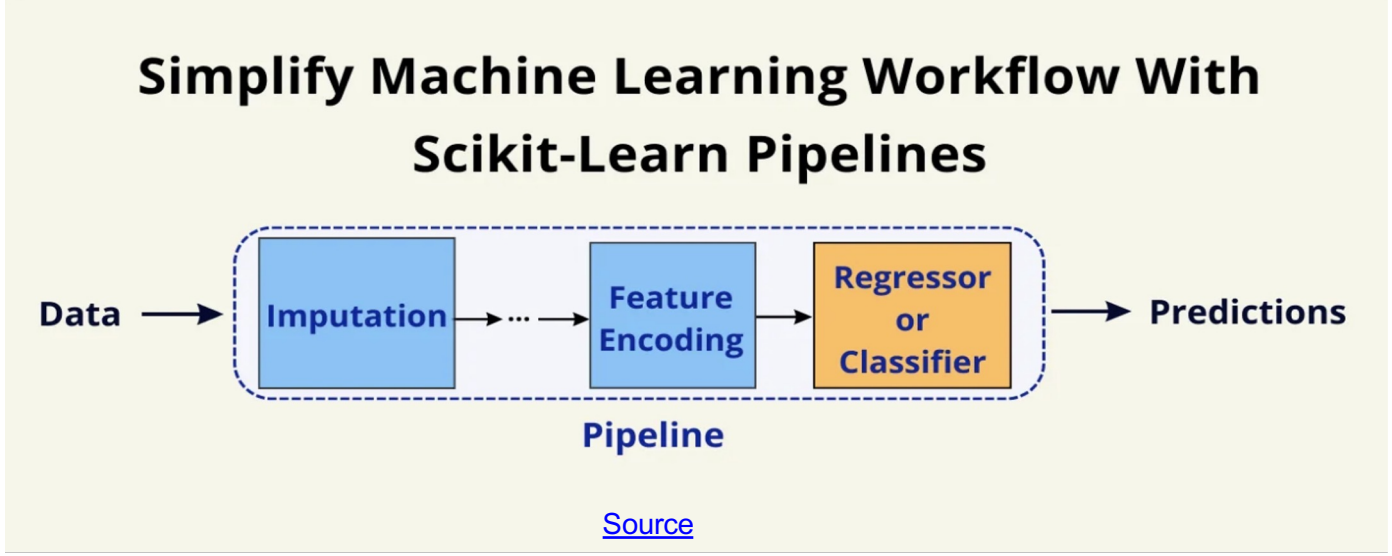


Source

Types of Imputers

1.Mean Imputation: Replacing missing values with the mean (average) of the non-missing values in a column.

2.Median Imputation: Replacing missing values with the median (middle value) of the non-missing values in a column.

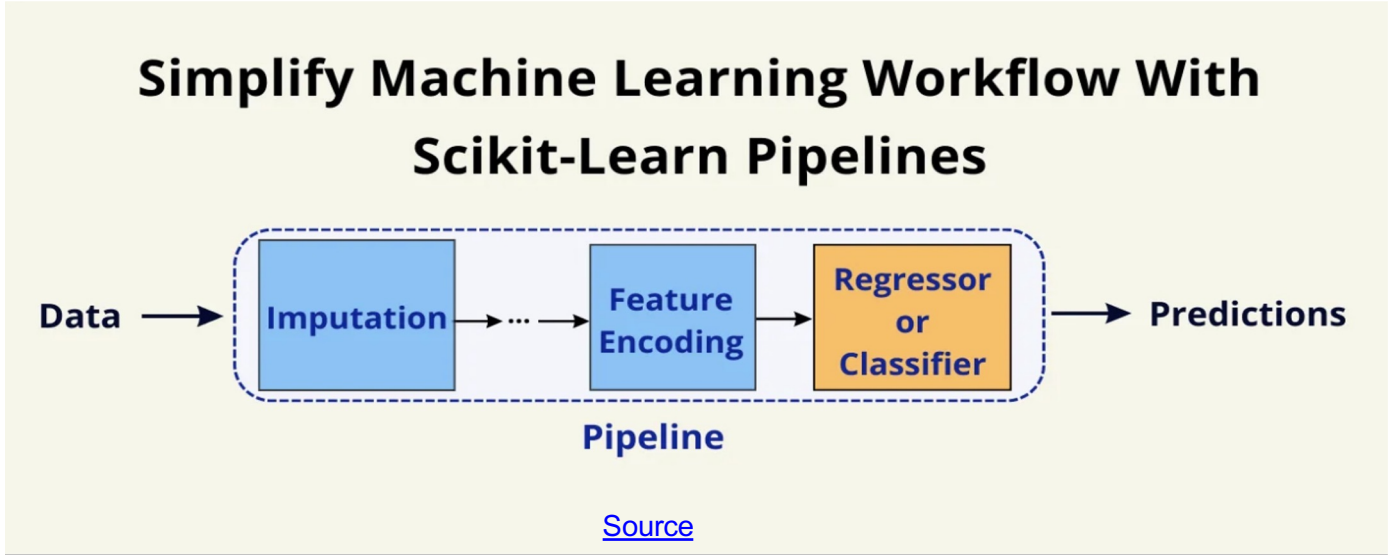


Source

Types of Imputers

3.Most Frequent Imputation: Replacing missing values with the most frequently occurring value in a column.

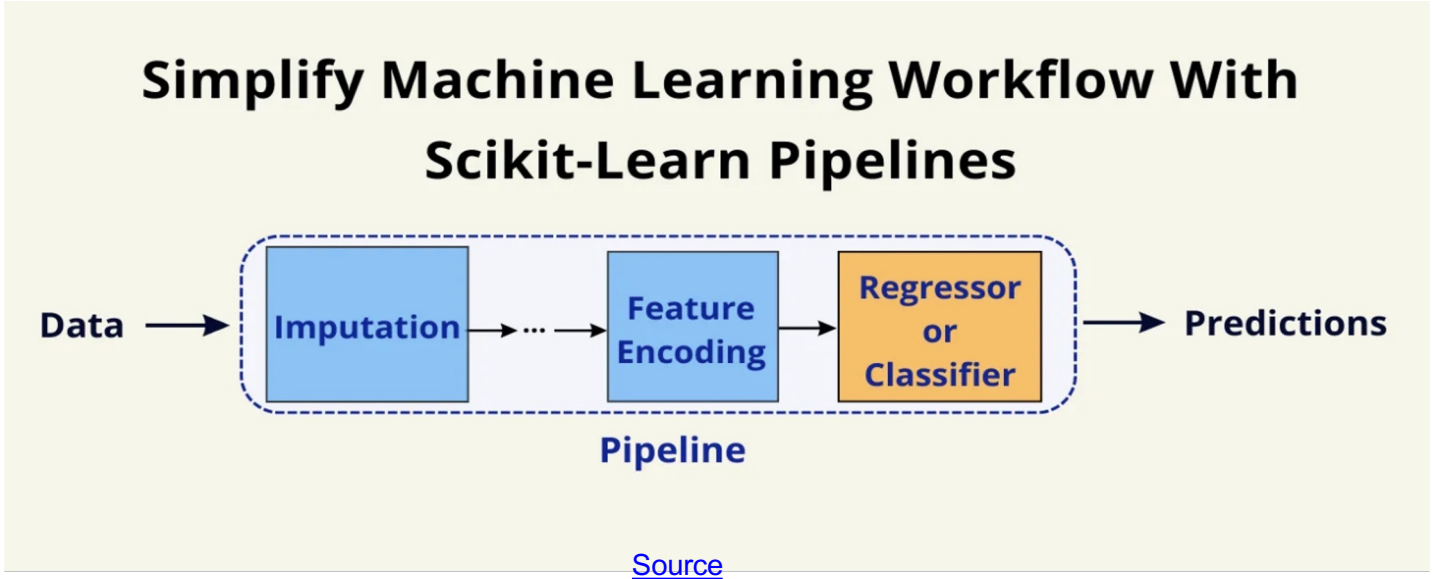
4.Constant Imputation: Replacing missing values with a constant value that you may specify.



Source

Encoder

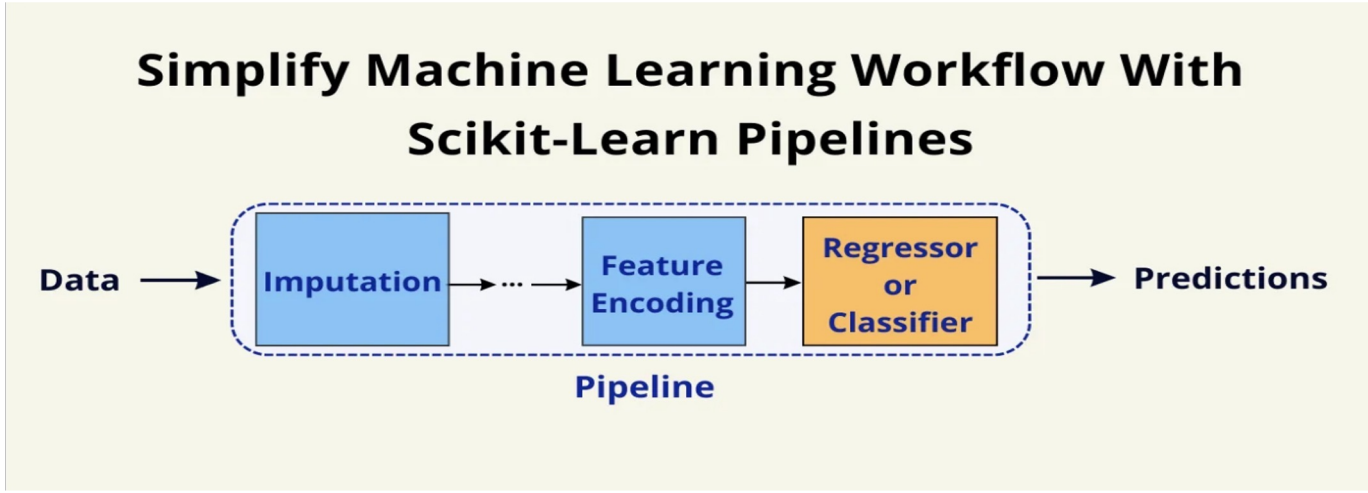
- An encoder is a tool used to convert categorical data into numerical format.
- Machine learning algorithms typically require numerical input. So, categorical data needs to be transformed into a format that can be understood by these algorithms.



Source

Types of Encoders:

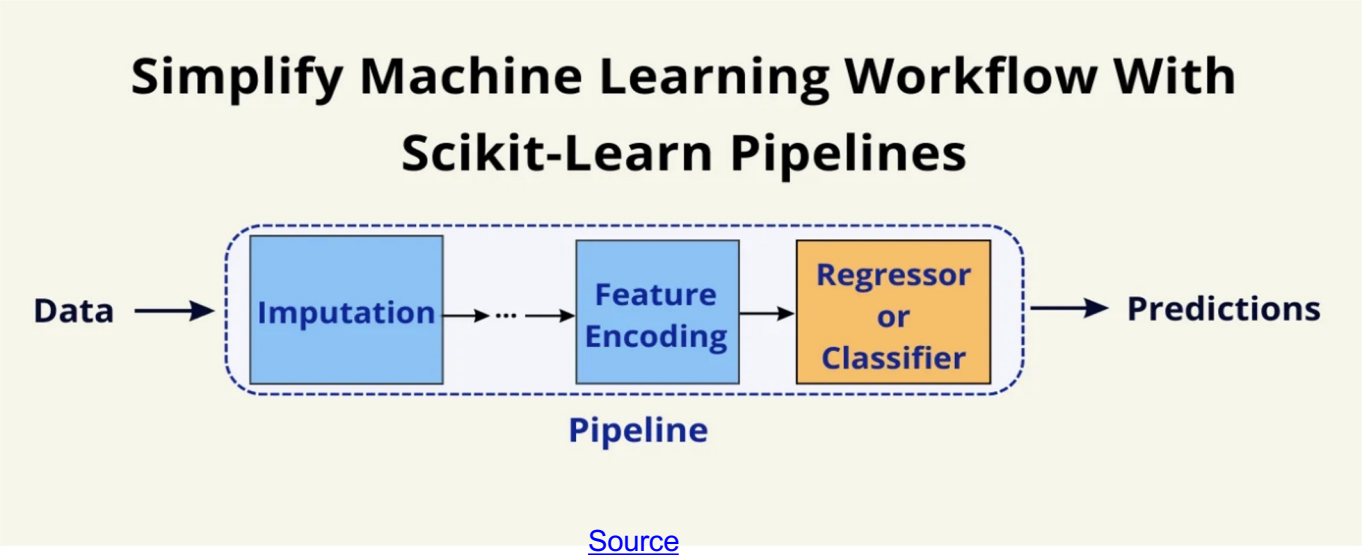
1.One-Hot Encoding: Converts categorical variables into a set of binary columns, each representing one possible category value.
For each sample, one of these columns will have a value of 1 (indicating the presence of the category) and the others will be 0.



Source

Types of Encoders:

2.Label Encoding: Converts each category value into a numerical label. This is simple but can sometimes introduce unintended ordinal relationships between categories.



Source

Why Use a Pipeline?

- **Streamline Workflow**: Combine multiple steps into one coherent process.
- **Reduce Errors**: Ensure that the same preprocessing is applied consistently.
- **Simplify Cross-Validation**: Easily apply the same transformations and model fitting across different folds.
- **Parameter Tuning**: Integrate hyperparameter tuning with preprocessing steps.

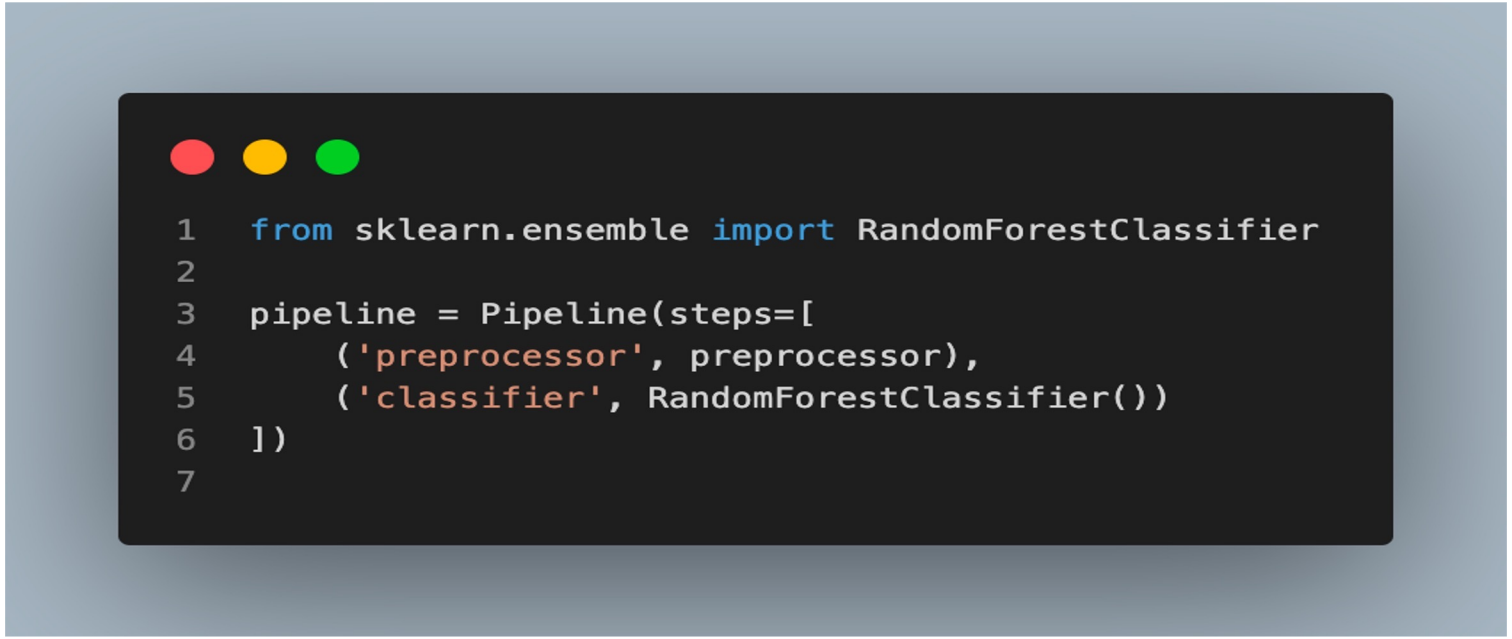
How does a pipeline work ?

1. **Define Steps**: Each step is a tuple consisting of a name and a transformer or estimator.
2. **ColumnTransformer**: This is used when you need to apply different preprocessing steps to different subsets of features.

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer

# Example column names (assuming numerical features only for simplicity)
numerical_features = data.feature_names
categorical_features = []  # Add categorical feature names if any

# Define transformers for numerical and categorical features
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Combine transformers using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ])

```

How does a pipeline work ?

1. **Define Steps**: Each step is a tuple consisting of a name and a transformer or estimator.
2. **ColumnTransformer**: This is used when you need to apply different preprocessing steps to different subsets of features.



```
from sklearn.ensemble import RandomForestClassifier

pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier())
])

```

Model Training (Recap)

Model training (Recap)

1. **Split Data:** split the data into train and test split.
2. **Train the Pipeline:** train the pipeline using fit method.

```
1  from sklearn.model_selection import train_test_split, cross_val_score
2  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
3  # Fit the pipeline on training data
4  pipeline.fit(X_train, y_train)
```


Model Validation and Evaluation (Recap)

Model validation and evaluation (Recap)

1.Validation: Validate your Pipeline Using Cross Validation from sklearn

2.Evaluation: Use the pipeline object for predictions and evaluations, ensuring that all preprocessing steps are applied consistently.

```
1  cv_scores = cross_val_score(pipeline, X_train, y_train, cv=5, scoring='accuracy')
2  print(f'Cross-validation scores: {cv_scores}')
3  print(f'Mean cross-validation score: {cv_scores.mean():.2f}')
4
```

Model validation and evaluation (Recap)

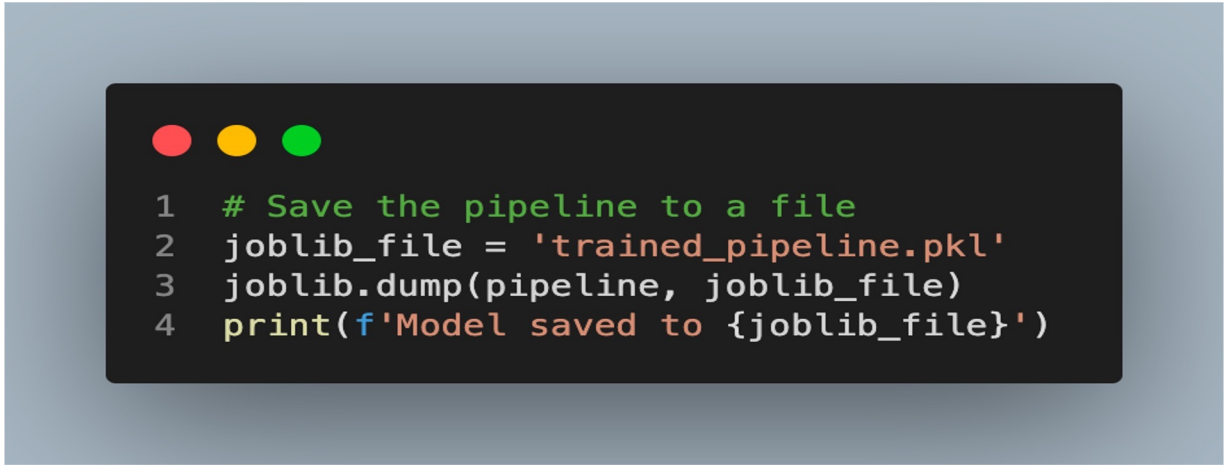
1.Validation: Validate your Pipeline Using Cross Validation from sklearn

2.Evaluation: Use the pipeline object for predictions and evaluations, ensuring that all preprocessing steps are applied consistently.

```
# Predict and evaluate on test data
y_pred = pipeline.predict(X_test)
accuracy = pipeline.score(X_test, y_test)
print(f'Test accuracy: {accuracy:.2f}')
```

Model Save

- In this step, we save the entire trained pipeline, which includes all preprocessing steps and the trained model, to a file.
- This allows us to preserve the state of the model and all its components so that we can reuse it later without needing to retrain or reapply preprocessing.



Load Model

- In this step, we load the previously saved pipeline from a file.
- This allows us to use the trained model and the preprocessing steps without needing to retrain the model or reapply preprocessing.
- We then use the loaded pipeline to make predictions and evaluate its performance.

```
# Load the pipeline from the file
loaded_pipeline = joblib.load(joblib_file)

# Use the loaded pipeline to make predictions
loaded_pipeline_predictions = loaded_pipeline.predict(X_test)
loaded_pipeline_accuracy = loaded_pipeline.score(X_test, y_test)
print(f'Loaded pipeline test accuracy: {loaded_pipeline_accuracy:.2f}')
```

Tutorial

MLOps_PipeLine_Tutorial.ipynb

Build API for ML Apps

What is an API?

An application programming interface is a way for two or more applications or components to communicate with each other.

It is a type of software interface, offering a service to other pieces of software.

Why is API important?

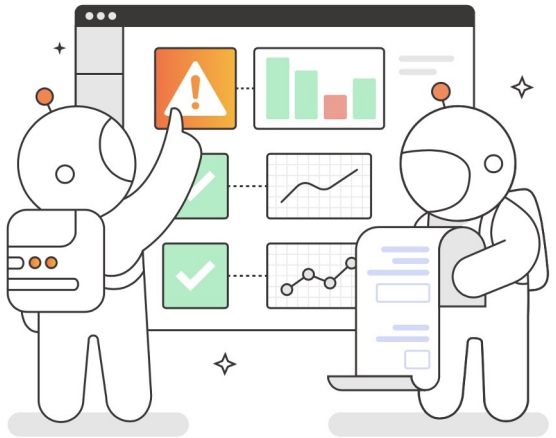
- **Interoperability:** APIs enable different applications (regardless of their technology) to interact with each other.
- **Reusability:** APIs promote code reuse, reducing development time and effort.
- **Abstraction:** APIs enables abstraction of a model or application by hiding its inner complexity and providing interface to interact with.

Types of APIs

- **RESTful APIs:** Representational State Transfer APIs are based on HTTP.
- **SOAP APIs:** Simple Object Access Protocol APIs use XML for message exchange.
- **GraphQL APIs:** A query language for APIs that allows clients to request only the data they need.

What is API testing

- API testing involves checking to make sure an API functions correctly.
- Developers can perform these tests by hand or use an API testing tool to automate the process.
- There are various kinds of API tests, each serving a unique purpose in making sure the API stays dependable.



Source

API Terminology (Request)

- **Endpoint:** The URL that the client sends a request to.
- **HTTP Method:** The type of action the client wants to perform (GET, POST, PUT, DELETE, etc.).
- **Headers:** Additional information sent with the request (e.g., content type).
- **Body:** Data sent with the request (for POST and PUT requests).

API Terminology (Response)

- **Status Code:** A three-digit code indicating the status of the request (e.g., 200 for success, 404 for not found).
- **Headers:** Additional information sent with the response (e.g., content type).
- **Body:** Data sent back to the client as a response.

Request Types:

- **GET:** Used in retrieving the data from the server.
- **POST:** Used in submitting data to the server to create a new resource.
- **PUT:** Used in updating data on the server.
- **DELETE:** Used in removing data from the server.

What is FastAPI?

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.

Creating Your First FastAPI App

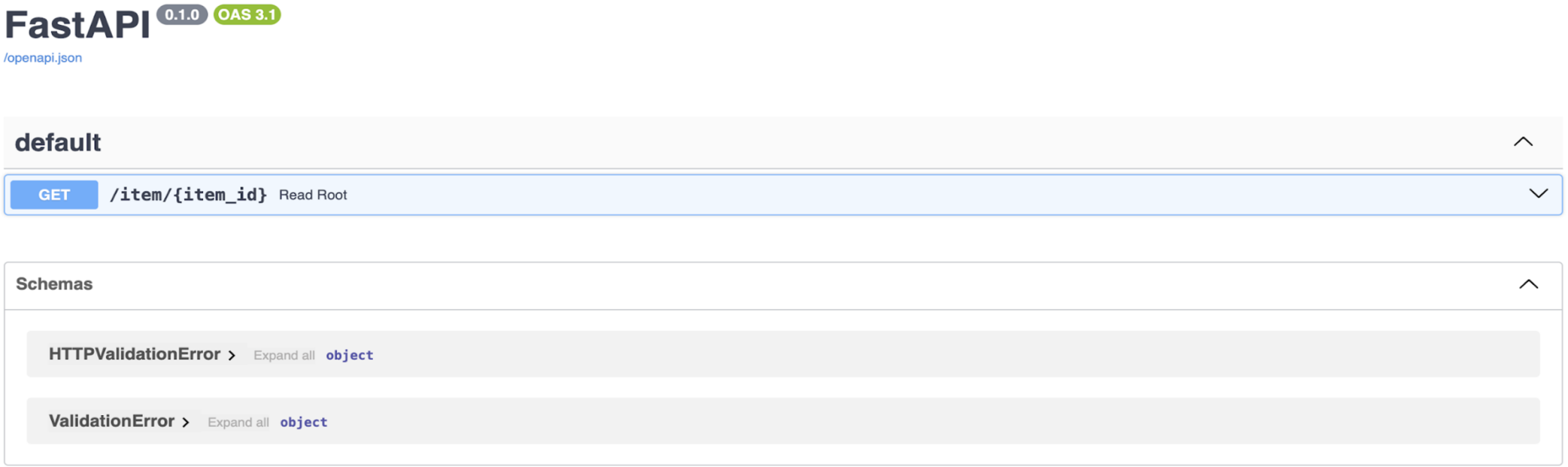
- To define your app routes, you will define it using decorators
  - @app.method(”path”)
- The function will handle the request with its parameters.
- Use async def and await for non-blocking code execution.
- Make a response to the received request.

BaseModel in FastAPI

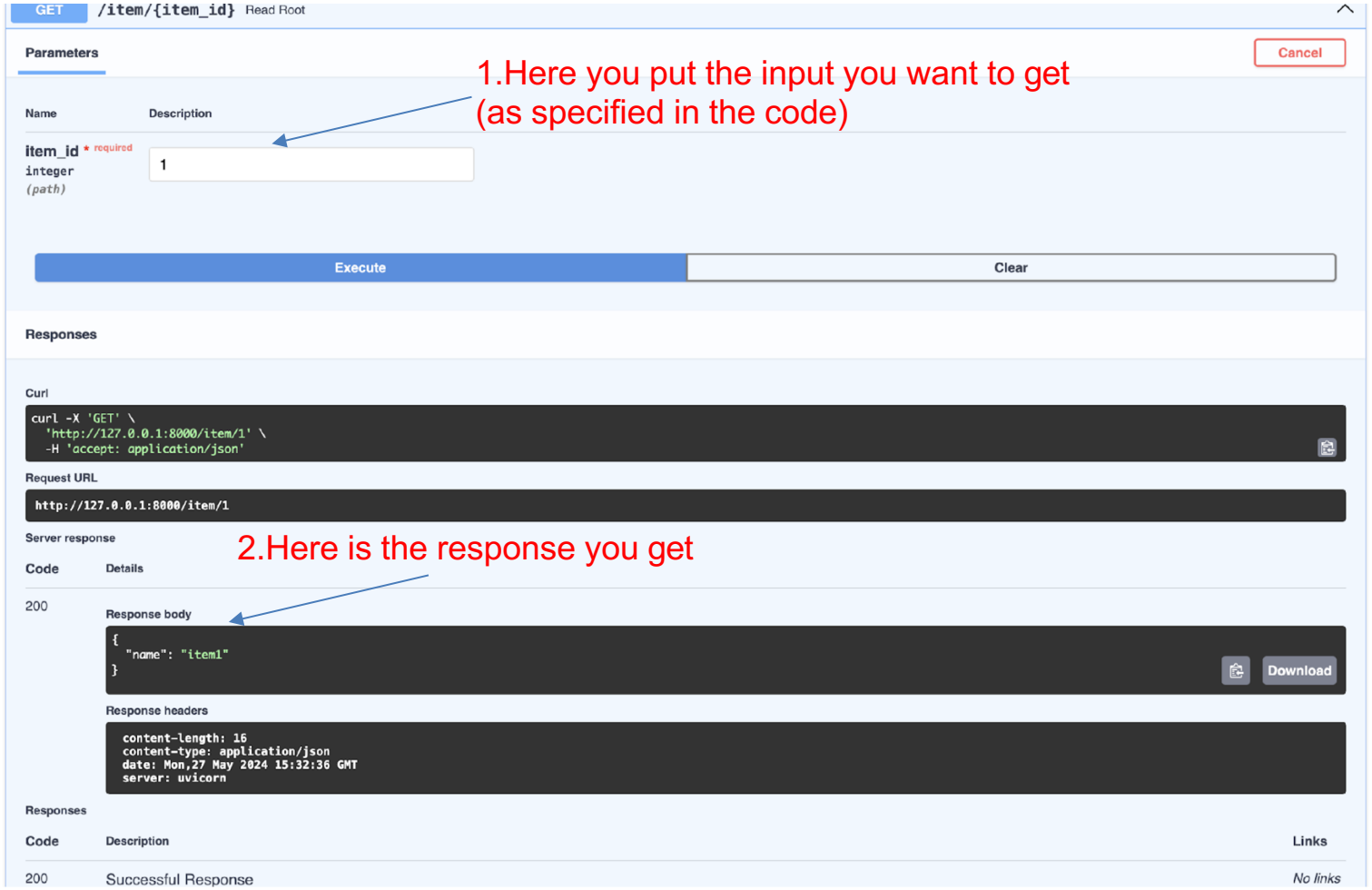
BaseModel is a special class provided by the Pydantic library, which is heavily used in FastAPI for data validation and serialization.

API Requests

- FastAPI uses Swagger UI to automatically create interactive API documentation.
- To view it, go to http://localhost:8000/docs and you'll see a page displaying all your endpoints, and methods.

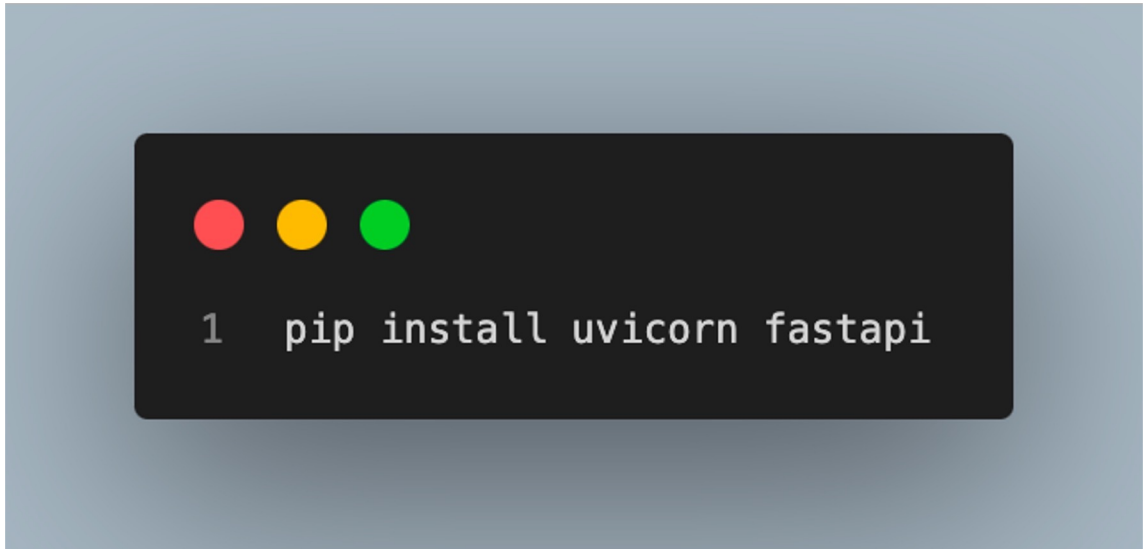


API Requests



Creating Your First FastAPI App

First, you need to install FastAPI and Uvicorn, which is an ASGI web server implementation for Python. You can do this by running the following command in the VS Code terminal:



Creating Your First FastAPI App

Create your *main.py* for the API and add the app:

Creating Your First FastAPI App

To run the code, use the following command on the VS Code terminal:

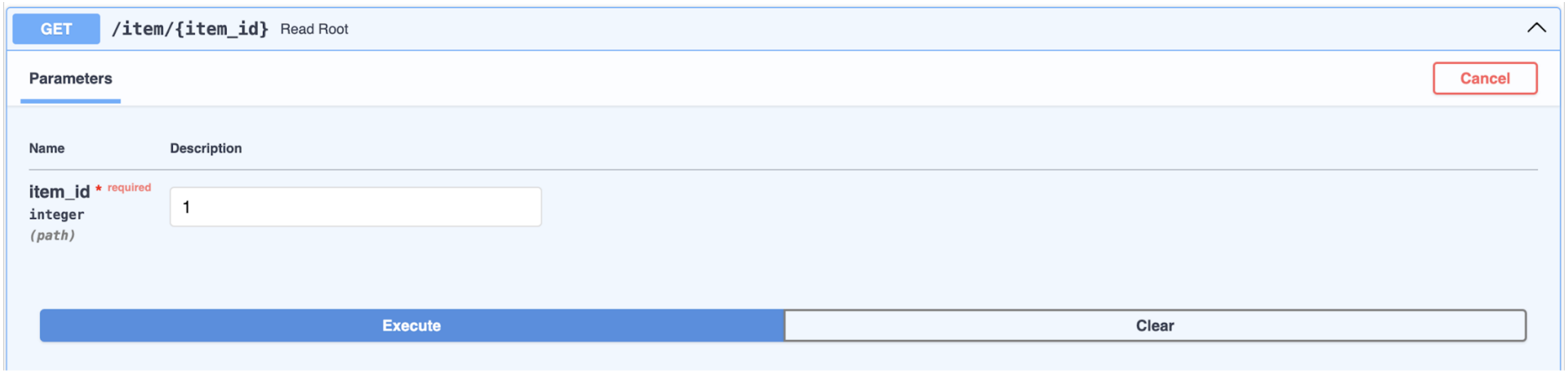
Creating Your First FastAPI App

GET Example Path Parameters:
We created a route to fetch items by their ID, which is passed as a parameter in the URL path and returned in the response.

```python
from fastapi import FastAPI
from fastapi.responses import JSONResponse
app = FastAPI()
items ={
    1: {"name": "item1"},
    2: {"name": "item2"},
    3: {"name": "item3"},
}
@app.get("/item/{item_id}")
async def read_root(item_id: int):
    return JSONResponse(content={"name": items[item_id]["name"]}, status_code=200)
```

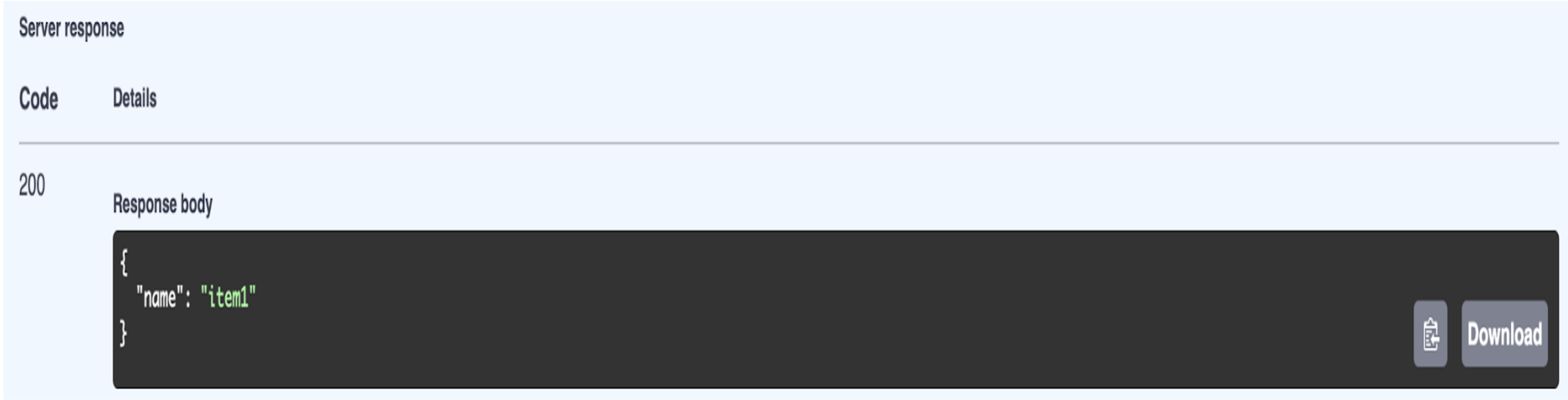

Creating Your First FastAPI App

GET Example Path Parameters output:
We passed the ID.



Creating Your First FastAPI App

GET Example Path Parameters output:
The item is returned successfully!



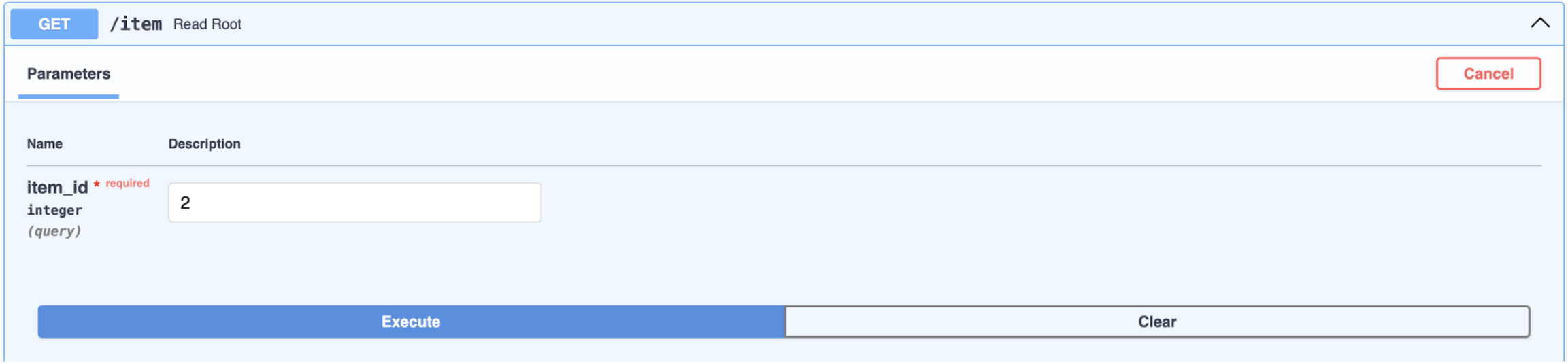
Creating Your First FastAPI App

GET Example Query Parameters:
We created a route to fetch items based on various query parameters, which are included in the URL as key-value pairs and used to filter the items returned in the response.

```python
from fastapi import FastAPI
from fastapi.responses import JSONResponse
app = FastAPI()
items ={
    1: {"name": "item1"},
    2: {"name": "item2"},
    3: {"name": "item3"},
}
@app.get("/item")
async def read_root(item_id: int):
    return JSONResponse(content={"name": items[item_id]["name"]}, status_code=200)
```

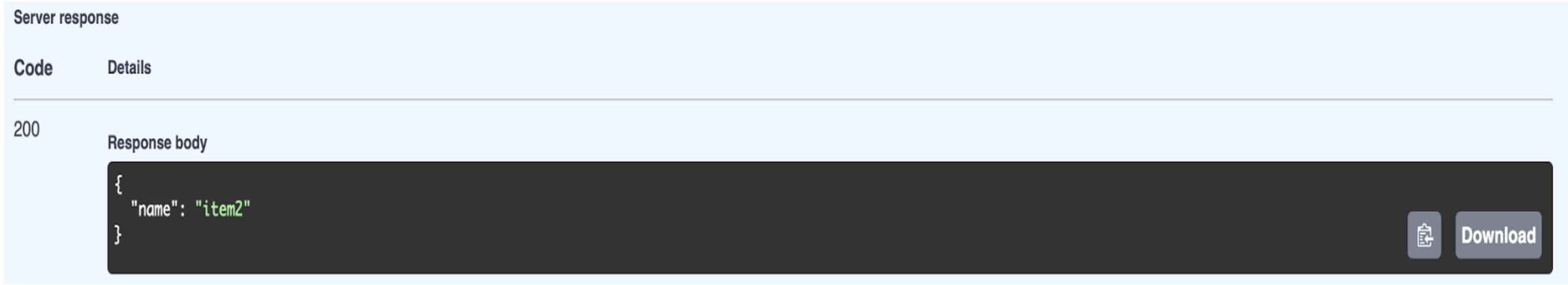

Creating Your First FastAPI App

GET Example Path Query Parameters:
We passed the ID.



Creating Your First FastAPI App

GET Example Path Query Parameters:
The item is returned successfully!



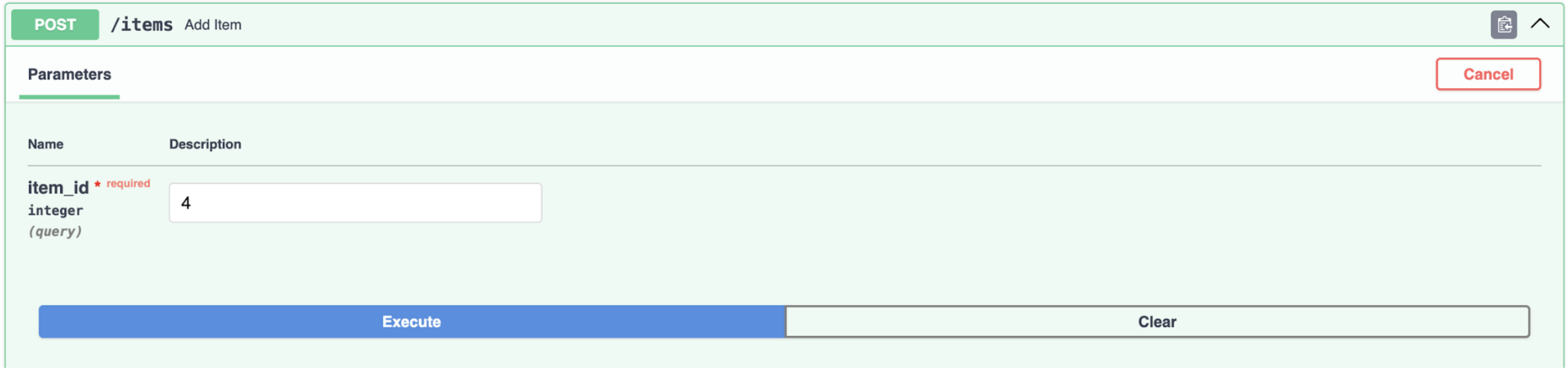
Creating Your First FastAPI App

POST Example:
We created a route to add a new item with a given ID. If the item ID already exists, it returns error with status code of 409. Else, it adds the item and returns the updated dictionary.

```python
from fastapi import FastAPI, HTTPException
from fastapi.responses import JSONResponse
app = FastAPI()
items = {
    1: {"name": "item1"},
    2: {"name": "item2"},
    3: {"name": "item3"},
}
@app.post("/items")
async def add_item(item_id: int):
    if item_id in items:
        raise JSONResponse(content={"message": "Item already exists"}, status_code=409)
    items[item_id] = {"name": f"item{item_id}"}
    return JSONResponse(content={"items": items}, status_code=200)
```

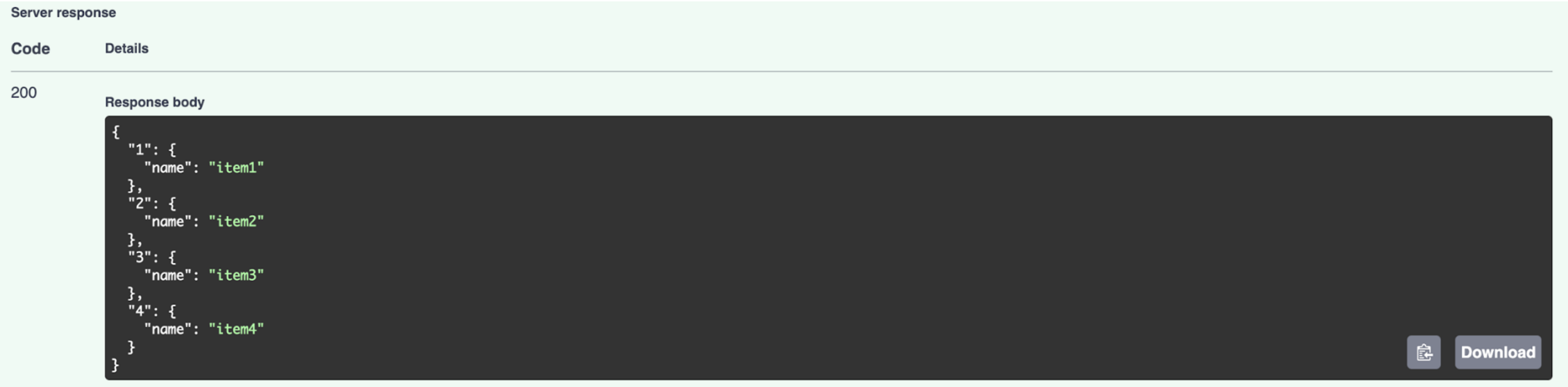

Creating Your First FastAPI App

POST Example output using Postman:
We passed the item ID.



Creating Your First FastAPI App

POST Example output using Postman:
The item is added successfully!



Creating Your First FastAPI App

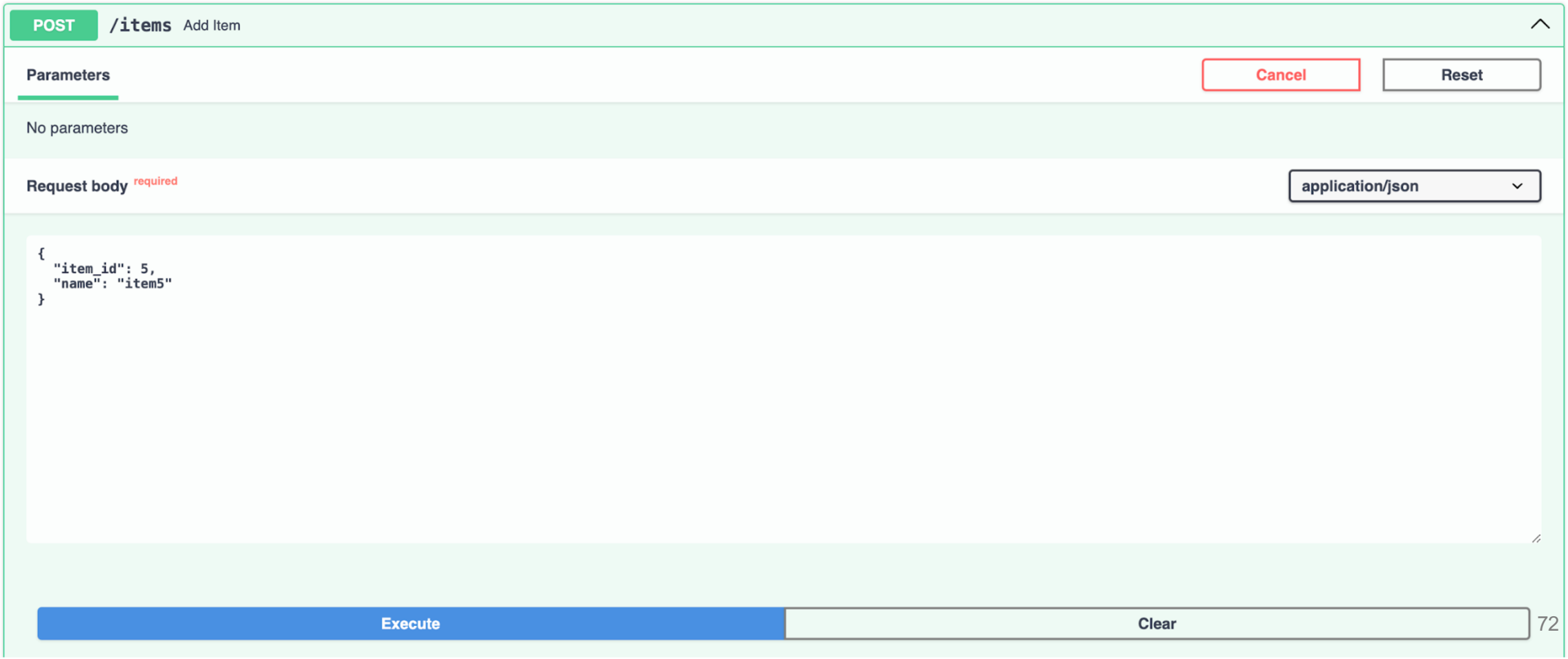
POST Example using BaseModel:

We create a route to add a new item with a but this time the item is received as a JSON using BaseModel.

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
app = FastAPI()
items = {
    1: {"name": "item1"},
    2: {"name": "item2"},
    3: {"name": "item3"},
}
class Item(BaseModel):
    item_id: int
    name: str
@app.post("/items")
async def add_item(item: Item):
    if item.item_id in items:
        raise HTTPException(status_code=409, detail="Item already exists")
    items[item.item_id] = {"name": item.name}
    return items
```


Creating Your First FastAPI App

POST Example output using Postman:
We passed the item ID and its name



Creating Your First FastAPI App

POST Example output using Postman:
The item is added successfully!

Server response

Code	Details
200	Response body

```
{
  "1": {
    "name": "item1"
  },
  "2": {
    "name": "item2"
  },
  "3": {
    "name": "item3"
  },
  "5": {
    "name": "item5"
  }
}
```

Download

Tutorial

Tutorial(IMPORTANT).pdf

Exercise

MLOps_lab.pdf

References:

- https://medium.com/geekculture/how-to-use-sklearn-pipelines-to-simplify-machine-learning-workflow-bde1cebb9fa2
- https://en.wikipedia.org/wiki/API
- https://konghq.com/blog/learning-center/different-api-types-and-use-cases
- https://fastapi.tiangolo.com/
- https://www.postman.com/api-platform/api-testing/