

Module 5: AST-3

TITLE: Model Serving through APIs

LEARNING OBJECTIVES:

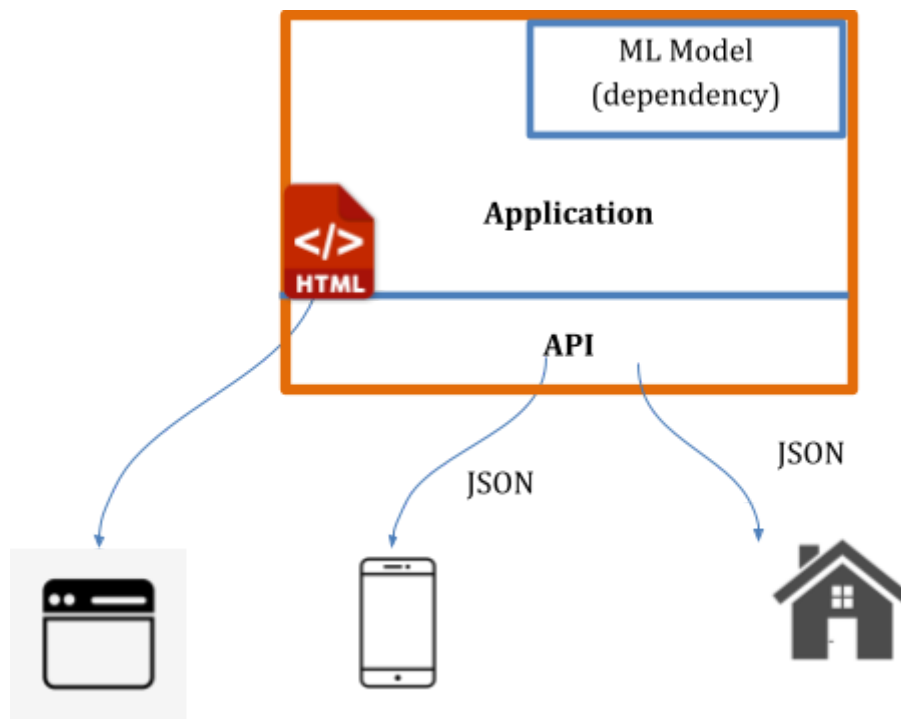
At the end of the experiment, you will be able to understand and build an API for the ML model.

You will be able to understand and implement the following aspects:

1. APIs, Status codes, and HTTP methods
2. Build a FastAPI for the ML model
3. Install packaged model that is consumed inside a FastAPI application

INTRODUCTION

Model Serving: There are different ways to deploy a model in a production environment: 1) Through API, and 2) Through application-mobile/web. API consumed via Web browsers (HTML), Mobile Devices, IoT, and other applications as shown in fig. below.



What is an API?

- **API:** Application Programming Interface (API) is a contract between an information provider and an information user that enables the communication between the two. Here information provider and user both are a kind of software and they don't have to be in the same language



- REST (Representative State Transfer) API is a type of contract. A set of rules that both parties adhere to.

REST-API has the following characteristics:

- Client-Server (like browser-web server)
- Has URI (like web server): Uniform resource identifier
- Stateless (doesn't remember your last call)

If we want our application to be available for other developers, creating an API acts as an intermediate connector. Developers will send HTTP requests to this API to consume the service. Users don't need the code or install dependencies. They only call this API through HTTP requests. It also simplifies integration, making it easier to integrate with third-party solutions. Before using an API, we must know the APIs, HTTP methods, and error codes.

HTTP methods & Status codes

HTTP methods	HTTP status code
GET: retrieve an existing resource (read only)	2xx: Successful operation
POST: Create a new resource/send information	3xx: Redirect
PUT: Update an existing resource	4xx: Client side error
PATCH: partially update an existing resource	5xx: Server side error
DELETE: Delete a resource	

FastAPI

It is a web development framework. A framework provides tools to handle & automate standard tasks when building web applications like Session management, Templating, Database access, and much more!

Python Web Frameworks: Flask (2011), Django (2006), FastAPI (2019) ...

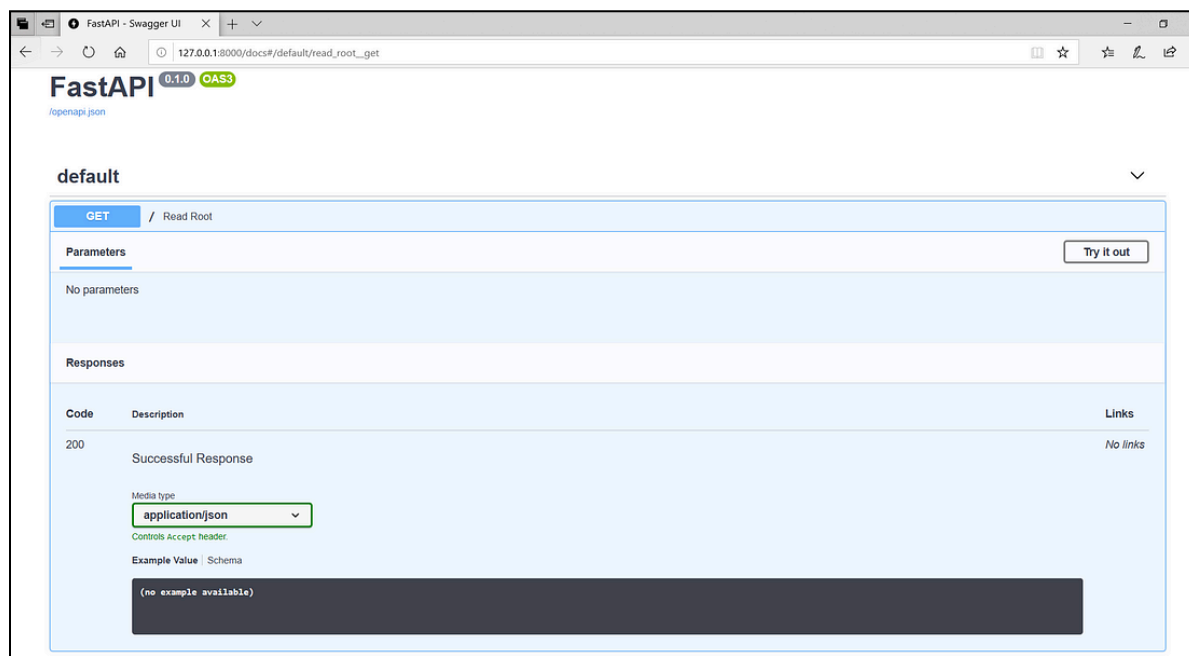
Features of FastAPI

- It is fast-leverage Python async capabilities
- Validation with type hints (python 3.6) and Pydantic: Helps in validation and automation, saves time & reduces bugs.
- Automatic Documentation
- Dependency Injection
- Much more!

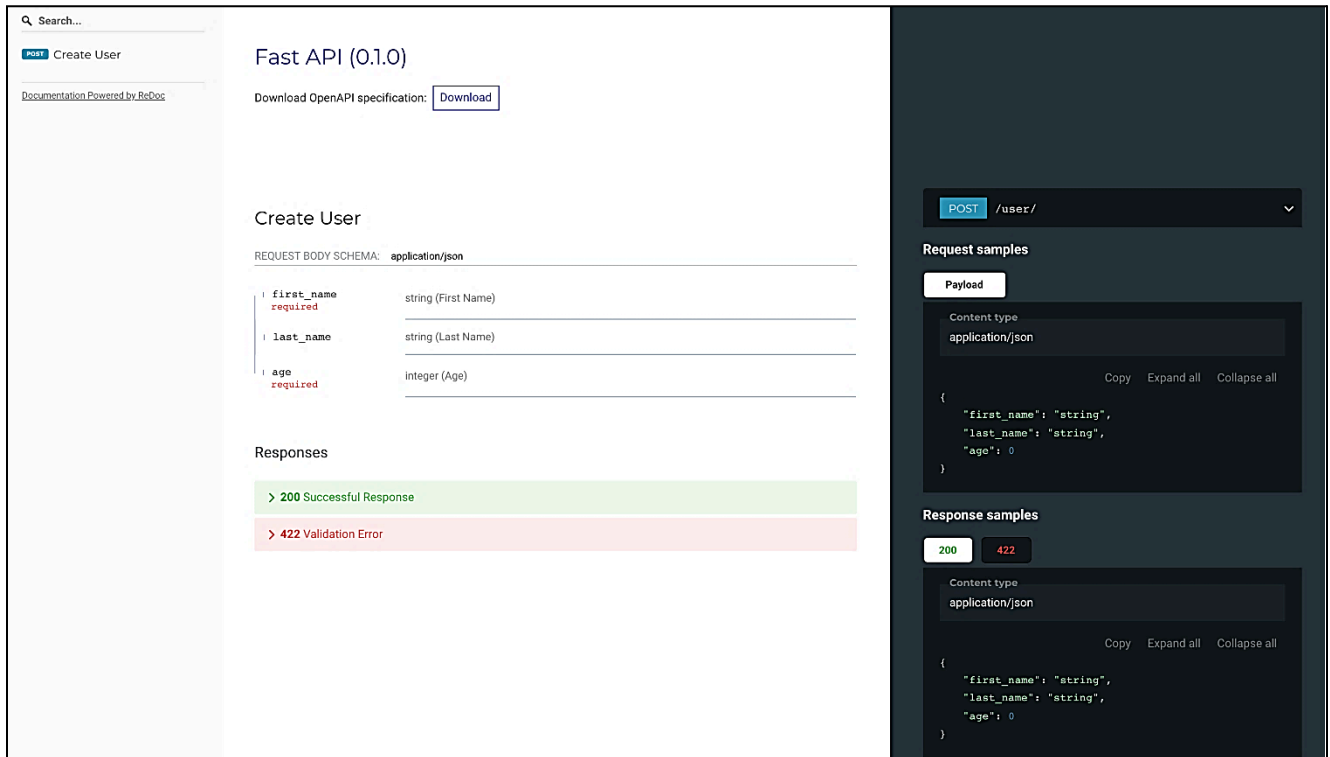
FastAPI Documentation

Given below are Swagger documentation that are automatically generated by FastAPI. Two different types of documents that we can access from localhost - docs & redoc, as shown below.

1. <http://localhost:8000/docs>



2. <http://localhost:8000/redoc>



Fast API (0.1.0)

Download OpenAPI specification: [Download](#)

Create User

REQUEST BODY SCHEMA: application/json

Field	Type
first_name (required)	string (First Name)
last_name (required)	string (Last Name)
age (required)	integer (Age)

Responses

- > 200 Successful Response
- > 422 Validation Error

Request samples

POST /user/

Payload

Content type: application/json

```
{
  "first_name": "string",
  "last_name": "string",
  "age": 0
}
```

Response samples

200 422

Content type: application/json

```
{
  "first_name": "string",
  "last_name": "string",
  "age": 0
}
```

Fast API Basics

Basic Function

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home():
    return {"Hello": "World"}

if __name__ == "__main__":
    uvicorn.run("hello_world_fastapi:app")
```

HTTP Methods

```
@app.get("/")
def home():
    return {"Hello": "GET"}

@app.post("/")
def home_post():
    return {"Hello": "POST"}
```

Explanation: Firstly, import **fastapi** and **uvicorn**. **Fast API** is developed on top of the **Uvicorn** library. The **‘app’** is initialized with the fast API class. The **‘get’** method is used to obtain the information in the app decorator. Also, a backslash indicates that it will work in the default local host URL. In the next step, a function **‘home’** is added to execute by API. In this case, it will execute the home function that returns the string – “Hello world”. Finally, **uvicorn.run** functions to execute the API with the name **“hello_world_fastapi: app”**. Different HTTP methods can be used with the **‘app’** decorator.

Query Parameters

```
@app.get("/employee")
def home(department: str):
    return {"department": department}
```

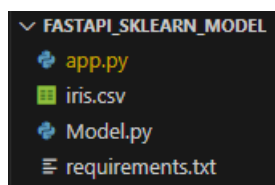
Path Parameters

```
@app.get("/employee/{id}")
def home(id: int):
    return {"id": id}
```

Explanation: The **query parameters** allow to send different values to a function. The **‘home’** function has a query parameter called **‘department’**, and this parameter will take different values based on the information that the user adds. The department parameter takes a value and then returns that value. This value will vary depending on the information the user sent to the function. Notice the **colon (:)** after the department followed by **str**. This is **data type validation** with **pydantic**. This function will validate that the department value data type is a string.

The **path parameters** are introduced where the path is specified. In this case, this API has an employee **‘id’** sub-path. The **‘id’** value will vary depending on the value inserted in the path. This **‘id’** parameter is specified between brackets, which is specific to path parameters. This idea will take different values depending on the value that the user passes to the path. For example, if we use: **‘employee/1’**, it would take the value of one. This value in turn will be passed to the home function where it will validate the data type.

Data Type Validation via Pydantic: It will be demonstrated with a full-fledged example given below. Example folder structure and modules:



```
# model.py
# Library imports
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from pydantic import BaseModel
import joblib

# Class which describes a single flower measurement
class IrisSpecies(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float

# Class for training the model and making predictions
class IrisModel:
    # Class constructor, loads the dataset and loads the model
    # if exists. If not, calls the _train_model method and
    # saves the model
    def __init__(self):
        self.df = pd.read_csv('iris.csv')
        self.model_fname_ = 'iris_model.pkl'
        try:
            self.model = joblib.load(self.model_fname_)
        except Exception as _:
            self.model = self._train_model()
            joblib.dump(self.model, self.model_fname_)

    # Perform model training using the RandomForest classifier
    def _train_model(self):
        X = self.df.drop('species', axis=1)
        y = self.df['species']
        rfc = RandomForestClassifier()
        model = rfc.fit(X, y)
        return model

    # Make a prediction based on the user-entered data
    # Returns the predicted species with its respective probability
    def predict_species(self, sepal_length, sepal_width, petal_length, petal_width):
        data_in = [[sepal_length, sepal_width, petal_length, petal_width]]
        prediction = self.model.predict(data_in)
        probability = self.model.predict_proba(data_in).max()
        return prediction[0], probability
```

```
# app.py
# Library imports
import uvicorn
from fastapi import FastAPI
from Model import IrisModel, IrisSpecies

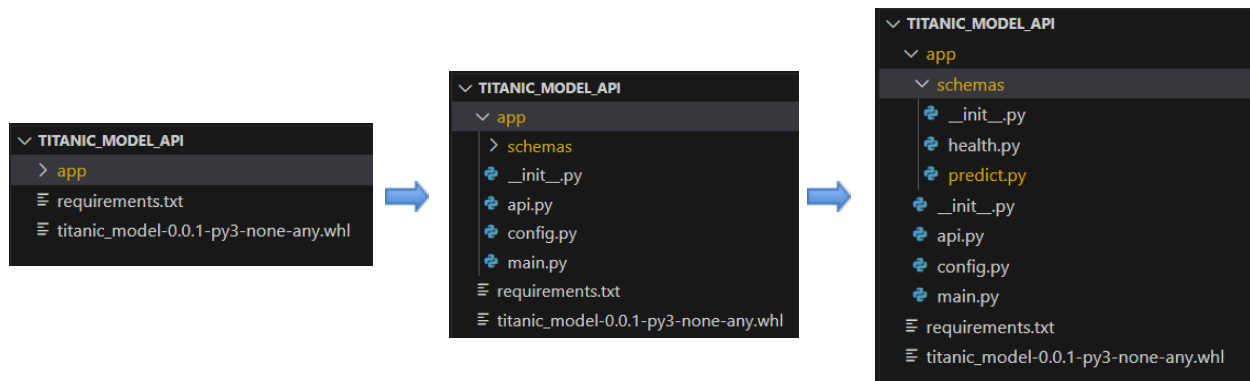
# Create app and model objects
app = FastAPI()
model = IrisModel()

@app.get('/')
def home():
    return {"Hello": "Welcome to the ML app."}
# Expose the prediction functionality, make a prediction from the passed
# JSON data and return the predicted flower species with the confidence
@app.post('/predict')
def predict_species(iris: IrisSpecies):
    data = iris.dict()
    prediction, probability = model.predict_species(
        data['sepal_length'], data['sepal_width'], data['petal_length'], data['petal_width']
    )
    return {
        'prediction': prediction,
        'probability': probability
    }

# Run the API with uvicorn
# Will run on http://127.0.0.1:8000
if __name__ == '__main__':
    uvicorn.run(app, host='127.0.0.1', port=8000)
```

```
#requirements.txt
uvicorn
fastapi
numpy
pandas
pydantic
scikit-learn
requests
joblib
```

Folder structure for installing the packaged model (from AST2) that is consumed inside the FastAPI application.



Understanding the functionality of each file

config.py

It is making use of **pydantic** and the **BaseSettings** to create the config class. The Settings class for specifying things like our project name etc. The class **Config** is specific to pydantic, and it just means that we can specify some options for our pydantic classes like they are case-sensitive. At last, the **settings** object is exported and this is imported throughout other modules.

main.py

We can see **config** and some imports from fastapi, chiefly the **FastAPI** class which is used to define the main **app**. The app uses a few different **routers** by calling a **root_router**. This is the way endpoints are defined in a fast API - the decorator's syntax (starting with **@**) with the router defining the user of the endpoint. In this case, it's just **the forward slash**.

A function is defined (**index**) which specifies an input **request** on call and this request object is imported from **fast API**. It gives us a few helper methods. And this is a very simple endpoint where we're defining a body and returning it as HTML. The HTML Response, again, is from Fast API. This gives the home endpoint where we see the '**Welcome to the API**'.

Another router i.e., **api_router** from the **API module** is imported. Note that both **api_router** and **root_router** have to specify using the **app.include_router** method which finally facilitates both the **root** and the **api**, which are both included in the app.

api.py

The **api_router=APIRouter()**, has been defined here, which is imported in that **main** module. It's the same syntax with the decorator for **health** endpoint, and for **predict** end point. In the case of the **health endpoint**, the response schema is defined, and returns that as a dictionary (**health.dict()**) . Under the hood fast api converts into JSON response. There are some fast API helper arguments like specifying that the status code for this endpoint by default is going to be 200.

Look at the **predict** end-point, which is more complex. The expected inputs **input_data** make use of a **schema**, where the **inputs are loaded** into a Pandas **DataFrame** and we use this fast API **Jsonable_encoder** which is imported from fastapi. This handles loading the **pydantic data into Jason** in the format that's expected by Pandas.

The '**make_prediction**' function has been called from the package. Recall that **make_prediction** returns the **dictionary**. Doing a check here - if there are errors, then it is going to **raise** an **HTTPException** and the client will receive a **status_code =400** as well as details of the error. If there are no errors, then we return the **Prediction results**. Need to replace the NumPy **np.nan** with **None** so that pedantic can work with them correctly.

Note: Router in fastapi helps to separate the code into multiple python files. If you are building an application or a web API, it's rarely the case that you can put everything on a single file. **FastAPI** provides a convenience tool i.e., router to structure your application while keeping all the flexibility.

Schemas directory

The '**health module**' inherits from this pedantic base model to define the **health class**, in a similar way to config. The type of each of the schema fields are specified using type hints which facilitates automatic validation. After starting the api, inside **api/v1/health**, these fields **name**, **API version** and **model** version correspond to those.

Inside the '**predict module**', the first schema is for **PredictionResults** and the second schema is **MultipleDataInputs** which uses the imported '**DataInputSchema**'. The multiple data inputs contain a nested field - '**inputs**'. The field inputs contain a list of each individual entry, which is a passenger for which we want to predict an output and the entry corresponds to the imported '**DataInputSchema**'.

Note that in the pydantic **Config** class, we're able to define an **example**. We have a single input example with a series of dummy values. The real value of specifying examples is that in documentation it is available to try out.

The super powerful things about fast API are its ability to pick up the **inputs** and **response models** of different endpoints. Based on a tiny bit of **config**, when a fast API app is instantiated by passing in the **open_api_url** (Fast API makes use of the open API standard which is baked deeply into the framework), we get out of the box documentation for any API where we specify response schemas. For example, if we navigate to **docs** and predict endpoints, we can see that the specified example & values are available to **try out**.

Uvicorn Web Server

It's important to have a dedicated Web Server which is specifically responsible for dealing with incoming requests and outgoing responses. So it's really important that we have one when we're for deploying to production, any of the standard Web servers will implement **Server Gateway Interface**. Historically, Python frameworks like Flask and Django have used what's known as the **Web server Gateway Interface (WSGI)**. This was introduced back in 2003 in PEP-333, and it was revised and PEP3333 to accommodate the introduction of Python 3.

Now, with the introduction of Python's **asyncio** Library, it was possible to go beyond the constraints of **WSGI**. As a result, the **Asynchronous Service Gateway Interface (ASGI)** was born.

UVICORN - the web server being used in the project is an implementation of this ASGI interface (doc: <https://www.uvicorn.org/>). The reason we're using this particular Web server is that it's recommended as a companion to FastAPI, although it's possible to use other ASGI servers as well.

It is able to make use of asynchronous applications and much faster with improved performance. Note the **uvicorn.run()** command which spins up the Web server and the web-app is able to start receiving requests and returning responses.

PROCEDURE

The following steps involve downloading the project folder and uploading it to VS code, creating a virtual environment within the project, installing necessary dependencies, and executing specific scripts. It allows us to effectively configure the project and execute it within the VS code environment.

Steps: Download the '**titanic_model_api**' folder and upload in your VS code and follow the steps below:

1. Create the virtual environment, activate it, and install the requirements.
2. Go inside the '**app**' directory and run the '**main.py**' file.
3. After successful run it should output
Application startup complete. Uvicorn running on http://0.0.0.0:8001
4. Now, go to <http://localhost:8001> or <http://127.0.0.1:8001> to open the fastapi app.