NYU FRE 7773 - Week 13

Machine Learning in Financial Engineering

Jacopo Tagliabue

Serving predictions

Machine Learning in Financial Engineering

Jacopo Tagliabue

Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.
2. Your model can be **trusted** by others, typically, other stakeholders, who may or may not be technical folks: you need to “make sure” the model behaved as designed before pushing it in front of end-users.
3. Predictions can be **consumed** by others, typically anybody with an internet connection: you need to expose your model as an endpoint which returns predictions when supplied with the appropriate parameters.

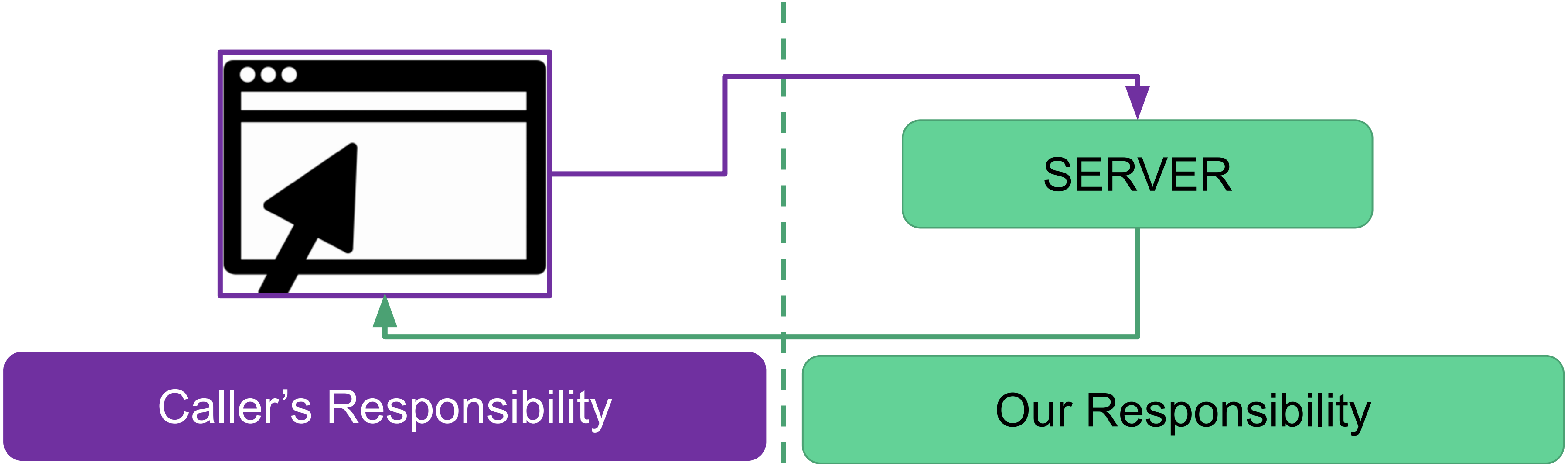
Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.
2. Your model can be **trusted** by others, typically, other stakeholders, who may or may not be technical folks: you need to “make sure” the model behaved as designed before pushing it in front of end-users.
3. Predictions can be **consumed** by others, typically anybody with an internet connection: you need to expose your model as an endpoint which returns predictions when supplied with the appropriate parameters.

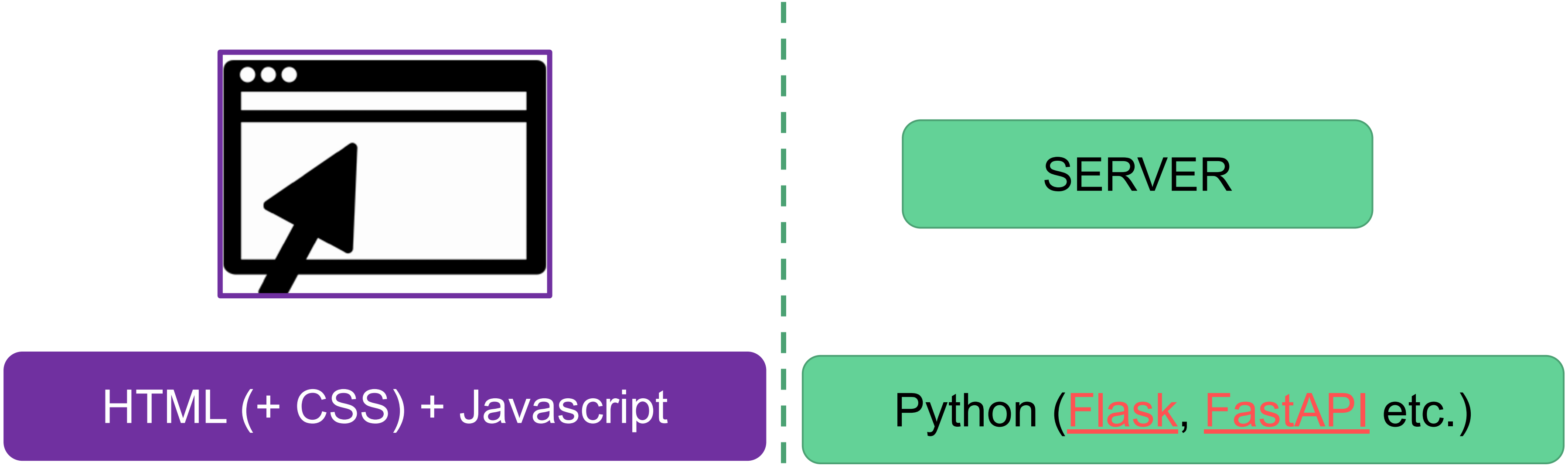
Part 3: Serving predictions

- If our model stays on our laptop, nobody will be able to use it!
- **Client-server architecture**: our model interacts with *many* remote clients through an API (also called “endpoint”) - we abstract away model code (and complexity) and expose a pure input-output interface: clients send us the input, we return a prediction.



Part 3: Serving predictions

- If our model stays on our laptop, nobody will be able to use it!
- **Client-server architecture**: our model interacts with *many* remote clients through an API (also called “endpoint”) - we abstract away model code (and complexity) and expose a pure input-output interface: clients send us the input, we return a prediction.



Show me first!

Intro to Flask applications

- We will be using Flask as a simple framework to serve model predictions after training
- Flask has several attractive features:
  - Helps with structuring both the front-end (the web page) and back-end (the endpoint)
  - Pure Python back-end
  - Minimal syntax for routing, GET / POST etc.

Step 1: prepare a web page

```html
<!DOCTYPE html>
<html>
<head>
        <title>{{ project }} app</title>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
</head>
<script type="text/javascript">

    $(function() {
        $('#predict').click(function() {
            event.preventDefault();
            var form_data = new FormData($('#myform')[0]);
            console.log(form_data);
            $.ajax({
                type: 'POST',
                url: '/',
                data: form_data,
                contentType: false,
                processData: false,
            }).done(function(data, textStatus, jqXHR){
                $('#result').text(data);
            }).fail(function(data){
                alert('error!');
            });
        });
    });

</script>
<body>
        <h1>{{ project }}</h1>
```

- Prepare a simple HTML page for users to interact with our endpoint.
    - It is not much different than the streamlit app we built before!
- Note that we use a simple Javascript function with jQuery to perform a POST request.

Step 2: prepare the Flask back-end application

```python
# We need to initialise the Flask object to run the flask app
# By assigning parameters as static folder name,templates folder name
app = Flask(__name__, static_folder='static', template_folder='templates')
```

- Initialize a Flask app in [app.py]()
  - Note the *templates* folder contains the HTML we created before!
- Make sure the app is started when we run “flask run”: the script will spin up a web server that will be ready to listen for incoming requests (from our HTML page, of course)

```python
if __name__=='__main__':
    # Run the Flask app to run the server
    app.run(debug=True)
```

Step 3: load the ML model in memory

```python
#### THIS IS GLOBAL, SO OBJECTS LIKE THE MODEL CAN BE RE-USED ACROSS REQUESTS ####

FLOW_NAME = 'MyRegressionFlow' # name of the target class that generated the model
# Set the metadata provider as the src folder in the project,
# which should contains /.metaflow
metadata('../src')
# Fetch currently configured metadata provider to check it's local!
print(get_metadata())

def get_latest_successful_run(flow_name: str):
    "Gets the latest successfull run."
    for r in Flow(flow_name).runs():
        if r.successful:
            return r

# get artifacts from latest run, using Metaflow Client API
latest_run = get_latest_successful_run(FLOW_NAME)
latest_model = latest_run.data.model
```

- As in all Python scripts, what is declared outside the functions is “global”.
  - In this context, this means that objects can live *across requests*: if the client ask for prediction 1 and then prediction 2, we do not need to reload the model, as it is already **in memory**!
- We want the model to be “global” as retrieving the model from Metaflow storage may be slow and expensive.

Step 4: defining our endpoint

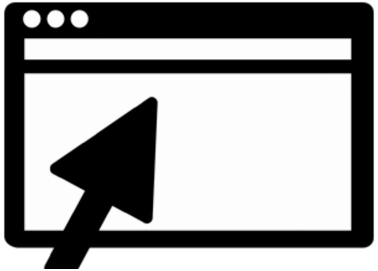
```python
@app.route('/',methods=['POST','GET'])
def main():

  # on GET we display the page
  if request.method=='GET':
    return render_template('index.html', project=FLOW_NAME)
  # on POST we make a prediction over the input text supplied by the user
  if request.method=='POST':
    # debug
    # print(request.form.keys())
    _x = request.form['_x']
    val = latest_model.predict([[float(_x)]])
    #  debug
    print(_x, val)
    # Returning the response to the client
    return "Predicted Y is {}".format(val)
```

- We use a decorator to define our route (empty in this case, but could be, say, “predict”).
  - If it was “predict”, our server would be listening for calls at **URL/predict**
- We distinguish between page load (GET) and request from a prediction (POST).

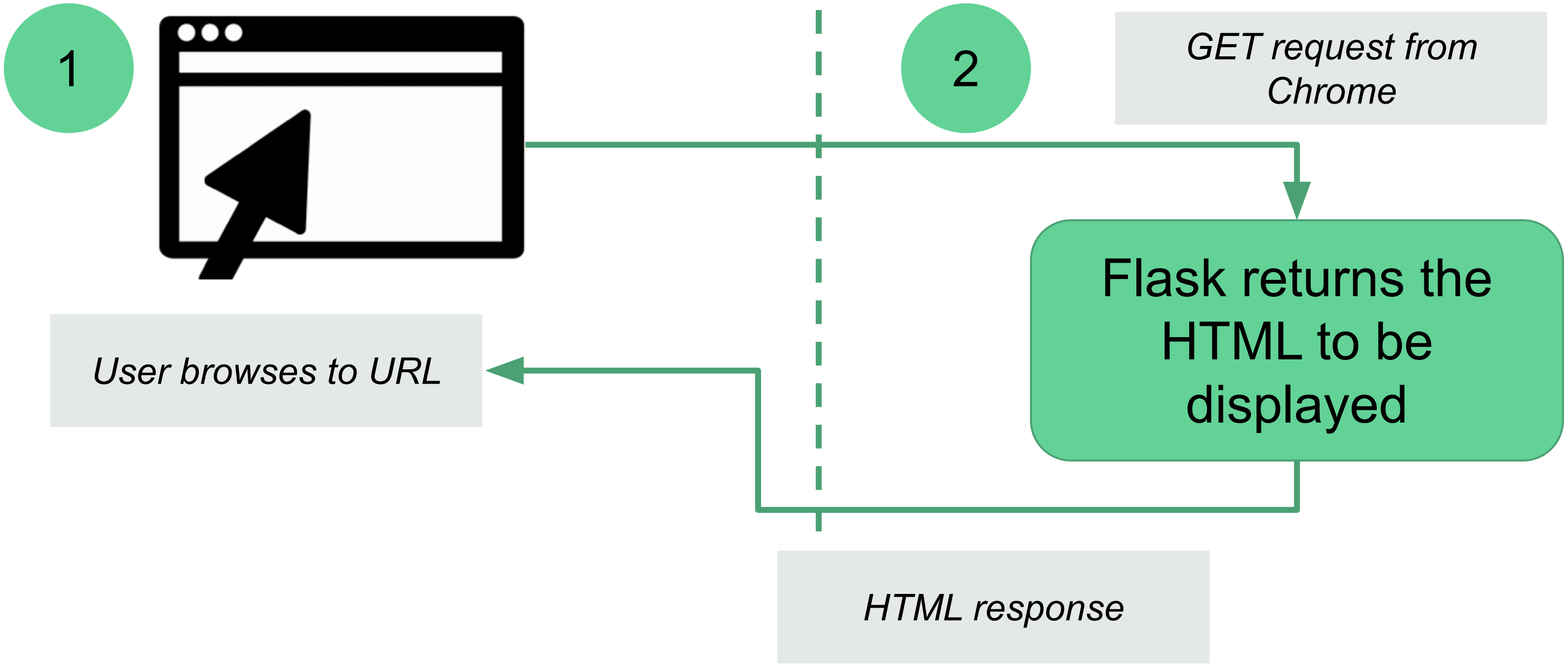
The full workflow

1



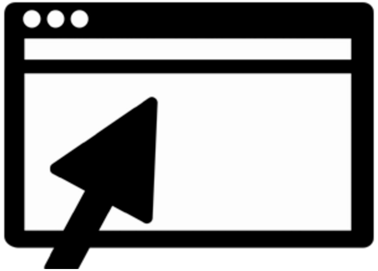
User browses to URL

The full workflow



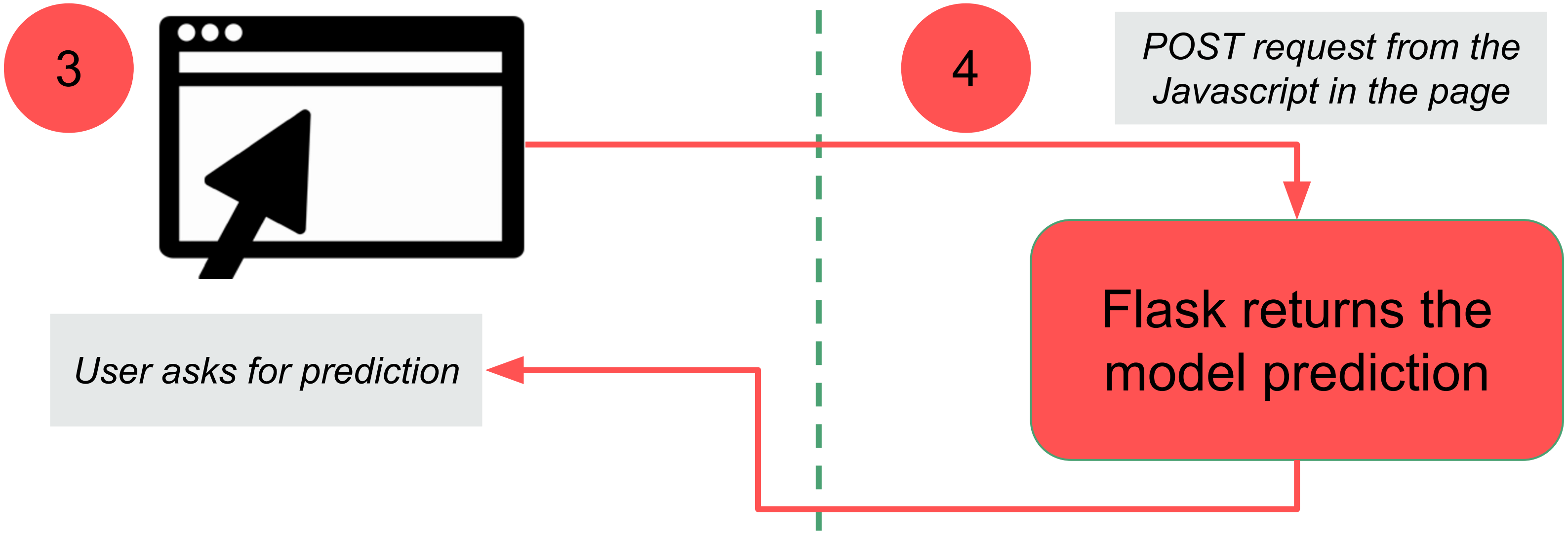
The full workflow

3



User asks for prediction

The full workflow



BONUS: Structuring the response

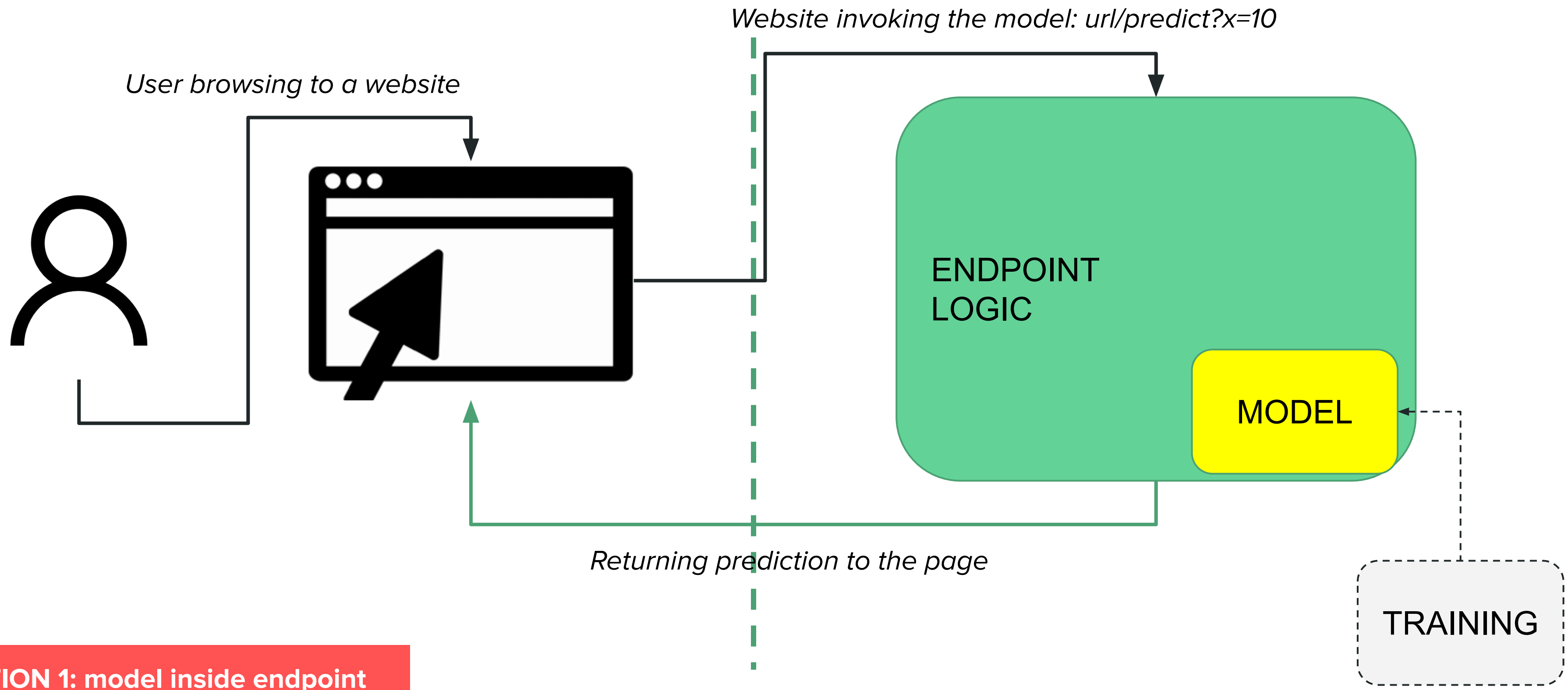
- Now **everybody** with the URL can use your awesome model!
- Can we make the response a bit clearer?

```
{
    "data": {
        "predictions": [167.068]
    },
    "metadata": {
        "eventId": "167b7129-cea1-4156-932f-f8d89c4b4066",
        "serverTimestamp": 163353256012,
        "time": 0.00022029876708984375
    }
}
```

This is the actual prediction from the model (why is it a list?)

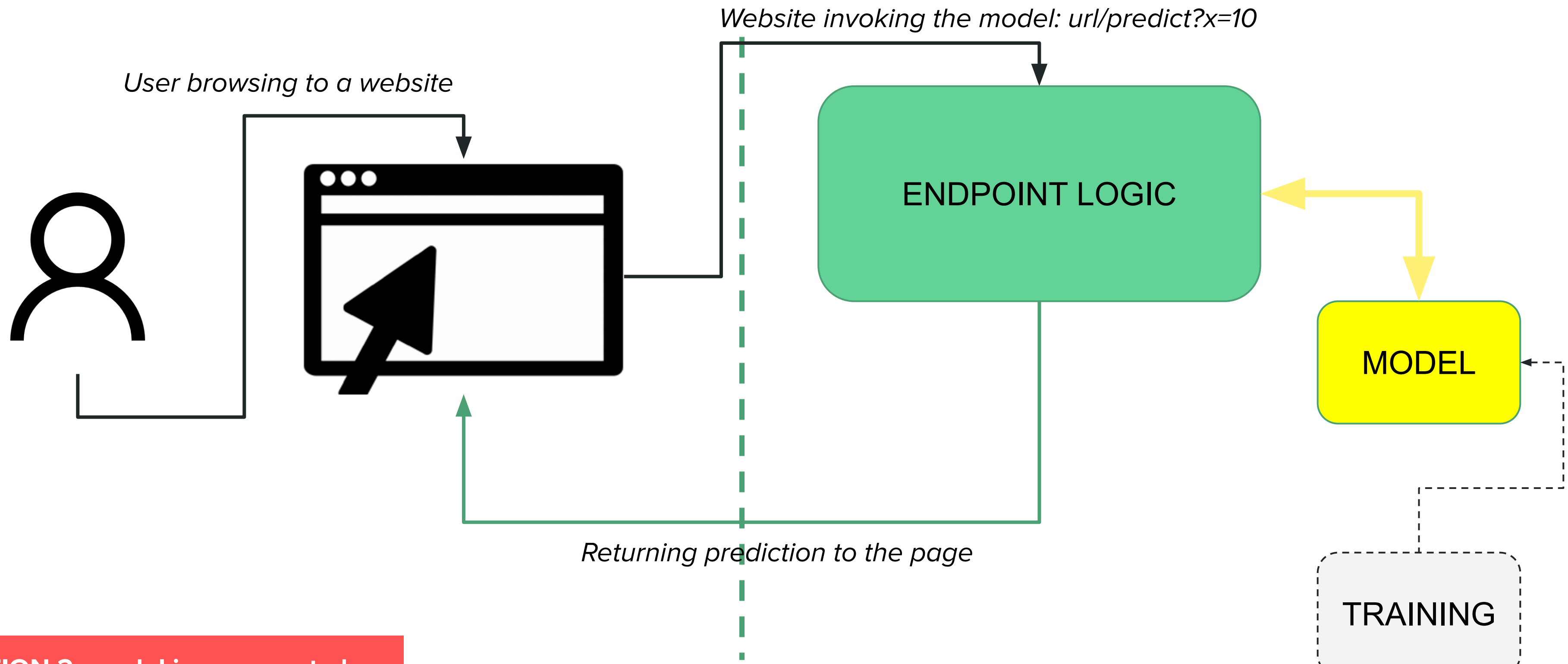
This is useful information about the call itself (debugging, monitoring, etc.)

Scenario 1: Endpoint with Model (ours)



OPTION 1: model inside endpoint

Scenario 2: Endpoint + Model



OPTION 2: model in a separate box

Flask... in the cloud

- There's 1M and one tutorials on how to use the very same tools (a Flask web app, a simple HTML + Javascript page) to port your app into the cloud.
- **BONUS points for your final demo if you can show your endpoint in the cloud**, either through an EC2 or Streamlit Cloud (**make sure to use the free resources / plan to avoid incurring in costs!**)

Alternative deployment scenarios

There is a ton of alternatives when it comes to *serving predictions* from the cloud, ranging from pure infrastructure to fully managed services. For example:

- You can deploy your model manually on a virtual machine, by installing Flask and run through screen (like they do here)
- You can deploy your model through a web app hosted by Elasticbeanstalk (like they do here)
- You can deploy your model through a web app hosted by Fargate (like they do here)
- You can deploy your model through Sagemaker, and expose it through a lambda (like we did in the 2021 repository)

After deployment: monitoring

We are not going to discuss monitoring, as we are not launching new apps in this course (for now!). However, after our model is live we need to:

- monitor how the pipeline is doing:
  - How is the new data coming in?
  - Does the model need re-training?
  - Is my new model better than the old one?
- check what users are doing with it!

After deployment: monitoring

We are not going to discuss monitoring, as we are not launching new apps in this course (for now!). However, after our model is live we need to:

- monitor how the pipeline is doing:
  - How is the new data coming in?
  - Does the model need re-training?
  - Is my new model better than the old one?
- check what users are doing with it!
  - You never know how people would use stuff!

The adventure never stops!

There is a ton of recent developments in the "MLOps" space (we do our small part as well in the community). If you want to know more, reach out!