**Serving the model via a REST API**

**Introduction**

**What is a REST API?**

* REST is a type of API
* API is the part of the server that receives requests and sends responses
* Follows the Client-Server architecture
* Client: Is the app itself, it’s the front-end
* Server: Gets and saves the data, it’s the back-end
* The communication happens using the HTTP protocol
* The Client calls the HTTP services from the Server
* REST builds these HTTP services, which includes Create, Read, Update and Delete data (aka CRUD Operations)
* We communicate with the Server using an endpoint, such as <http://cars.com/api/customers>
* The type of the HTTP request (methods) determines the kind of the operation, which includes:
  + GET (for getting/reading data)
  + POST (for creating data)
  + PUT (for updating data)
  + DELETE (for deleting data)
* In a nutshell: We expose our resources, such as costumers, using a simple address and enable various operations around them, such as creating or updating, using standard HTTP methods

**Flask Overview**

* Flask is a web framework, meaning that it provides tools, libraries and technologies that allow you to build a web application
* Microservices is an architecture that structures an application as a collection of services that are:
  + Highly maintainable and testable
  + Loosely coupled
  + Independently deployable
  + Organized around business capabilities
  + Owned by a small team

**Creating the API Skeleton**

* Show that ml\_api directory was added (which will be a package in the future)
  + That’s going to be used to separate the *api* from the *model*
* Show ml\_api\api\requirments
  + Each package will have its own requirements
* Show running pip install -r packages\ml\_api\requirements.txt
* Show ml\_api\run.py file, which is the entry point to start Flask
* Show ml\_api\api\app.py file
  + This is where we create the app, using the Factory design pattern
  + The *register\_blueprint* is creating one endpoint, which lives in the controller
* Show ml\_api\api\controller.py
  + It is a health end point which occurs in a HTTP GET request and returns ‘ok’
* Run *set FLASK\_APP=run.py* which tells Flask what the entry point is to start up our API

**Adding Config and Logging**

* Show ml\_api\api\config.py to set up ours handlers
* Show ml\_api\api\app.py
  + from\_object sets the *config* settings into our Flask app
  + \_logger.debug for logging
* Show ml\_api\tests\conftest, which is used by pytest to create setup and tear down functions
  + The fixtures can be passed in as arguments and then return values that are immediately useful
  + The fixtures are creating test instances of our Flask app, which then uses the Flask app*\_context*
* Show ml\_api\tests\test\_controller.py
* Run *pytest packages\ml\_api\tests*
* Show ml\_api\api\controller.py
* Show ml\_api\tests\run.py

**Adding the Prediction Endpoint**

* Show ml\_api\api\controller.py
  + We created a new endpoint (‘/v1/predict/regression’), which responds to a POST HTTP request
  + /v1/ is a good practice when creating APIs (which is unrelated to the version)
  + We are importing make\_prediction
  + We want to make our API as lightweight as possible
* Show ml\_api\tests\test\_controller.py
  + The model package contains everything needed to test it as well
* Run *pytest packages\ml\_api\tests*

**Adding a Version Endpoint**

* Show ml\_api\VERSION
* Show ml\_api\api\\_\_init\_\_.py
* Show ml\_api\api\controller.py
* Show ml\_api\tests\test\_controller.py
* Run the tests *pytest packages\ml\_api\tests*
* Run the service *python packages\ml\_api\run.py*

**API Schema Validation**

* Show ml\_api\requirements.txt
  + Marshmallow library is a Python library for doing objects serialization and validation
* Show ml\_api\api\controller.py
* Show ml\_api\api\valdiation.py
  + Marshmallow schemas
    - Schema is a representation of data, here it represents the inputs that we’ll be receiving to the API (it defines our API contract)
    - Work closely with whoever created the model to make sure that the fields are correct
    - Schema.load(input\_data) checks if the fields are formatted as expected
    - exc.messages is a dictionary where the keys are the rows where there were erros
    - can be used to:
      * Validate input data
      * Deserialize input data to app-level objects
      * Serialize app-level objects to primitive Python types. The serialized objects can then be rendered to standard formats such as JSON for use in an HTTP API
    - Many should be set to True if obj is a collection so that the object will be serialized to a list
  + No Python variables can start with numbers
* Show ml\_api\tests\test\_validation.py
* Run *pytest packages\ml\_api\tests*