

Create Django Proxy Services Of Backend APIs



Estimated time needed: 120 minutes

In previous labs, you created car models and car-made Django models residing in a local SQLite repository. You are also provided with dealer and review models Mongo DB served by express API end points.

Now, you need to integrate those models and services to manage all entities such as dealers and reviews.

To integrate external dealer and review data, you need to call the API end points from the Django app and process the API results in Django views. Such Django views can be seen as proxy services to the end user because they fetch data from external resources per users' requests.

In this lab, you will create such Django views as proxy services.

Run the Mongo Server

The backend Mongo Express server needs to be up and running in one of the terminals in the lab environment. At this stage, the server code will have all the end points implemented already.

1. Open a new Terminal.
2. Git clone your repository with all the changes you have made in the previous tasks.
3. Change to the database directory.

```
cd /home/project/xrwvm-fullstack_developer_capstone/server/database
```

4. Build the nodeapp.

```
docker build . -t nodeapp
```

5. Run the following command to start the server.

```
docker-compose up
```

6. Keep the server running in this terminal. You will need it for doing the rest of the lab.

7. Click the Backend button below, copy the URL in the address bar.

Backend

Note: If the button doesn't work, launch the application on Port 3030 to obtain the URL.

8. Open `djangoapp/.env` and replace the `your backend url` with the URL of your backend you copied earlier in the notepad in the previous step.

Make sure that the `/` at the end is not copied.

```
backend_url =your backend url
```

Please refer to [the lab](#) if required.

Environment setup

1. Open another new terminal.

2. Run the following to set up the django environment.

```
cd /home/project/xrwvm-fullstack_developer_capstone/server
pip install virtualenv
virtualenv djangoenv
source djangoenv/bin/activate
```

3. Install the required packages by running the following command.

```
python3 -m pip install -U -r requirements.txt
```

4. Run the following command to perform models migration.

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Create function to interact with backend

In the previous lab, you would have created a API endpoints to `fetchReviews` and `fetchDealers`. Now implement a method to access these from the Django app.

There are many ways to make HTTP requests in Django. Here we use a very popular and easy-to-use Python library called `requests`.

1. Open `djangoapp/restapis.py` and Uncomment the following import at the top of the file:

```
import requests
```

2. Add a following `get_request` method, as given below.

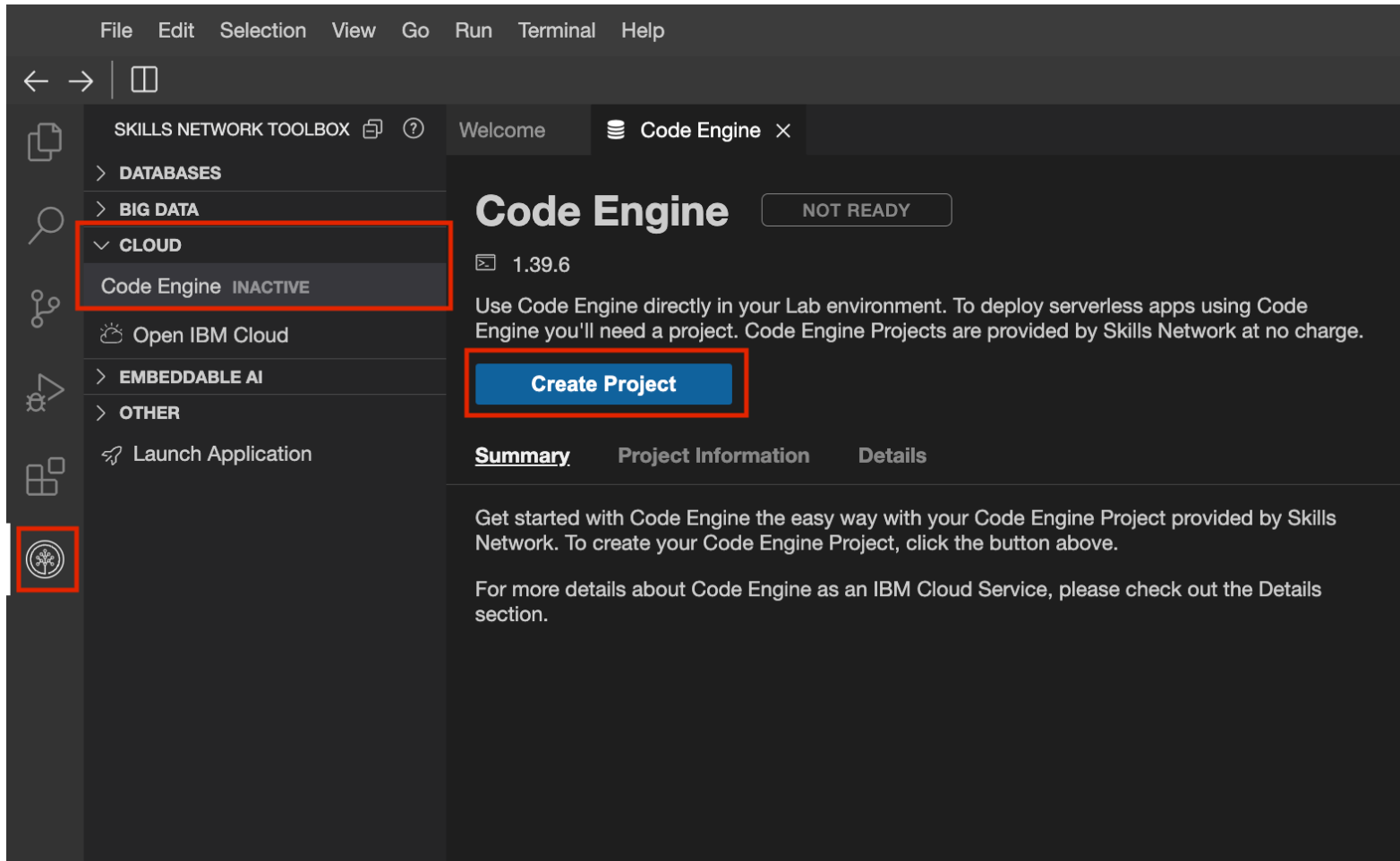
```
def get_request(endpoint, **kwargs):
    params = ""
    if(kwargs):
        for key,value in kwargs.items():
            params=params+key+"="+value+"&"
    request_url = backend_url+endpoint+"?" +params
    print("GET from {}".format(request_url))
    try:
        # Call get method of requests library with URL and parameters
        response = requests.get(request_url)
        return response.json()
    except:
        # If any error occurs
        print("Network exception occurred")
```

The `get_request` method has two arguments, the endpoint to be requested, and a Python keyword arguments representing all URL parameters to be associated with the get call.

This function calls `GET` method in `requests` library with a URL and any URL parameters such as `dealerId`.

Start the Code Engine

1. Start code engine by creating a project.



The screenshot displays the Skills Network IDE interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The left sidebar, titled 'SKILLS NETWORK TOOLBOX', contains categories: DATABASES, BIG DATA, CLOUD (expanded), and EMBEDDABLE AI. Under CLOUD, 'Code Engine INACTIVE' is highlighted with a red box. Below it is 'Open IBM Cloud'. Under EMBEDDABLE AI, 'Launch Application' is visible. A red box highlights the 'Code Engine' icon in the bottom-left corner of the sidebar. The main panel shows the 'Code Engine' setup. It has a 'Welcome' tab and a 'Code Engine' tab. The 'Code Engine' tab displays 'NOT READY' and version '1.39.6'. A red box highlights the 'Create Project' button. Below the button are tabs for 'Summary', 'Project Information', and 'Details'. The 'Summary' tab is active, showing instructions to get started with Code Engine and a link to the Details section.

File Edit Selection View Go Run Terminal Help

SKILLS NETWORK TOOLBOX Welcome Code Engine X

> DATABASES

> BIG DATA

✓ CLOUD

Code Engine INACTIVE

Open IBM Cloud

> EMBEDDABLE AI

> OTHER

Launch Application

Code Engine

NOT READY

1.39.6

Use Code Engine directly in your Lab environment. To deploy serverless apps using Code Engine you'll need a project. Code Engine Projects are provided by Skills Network at no charge.

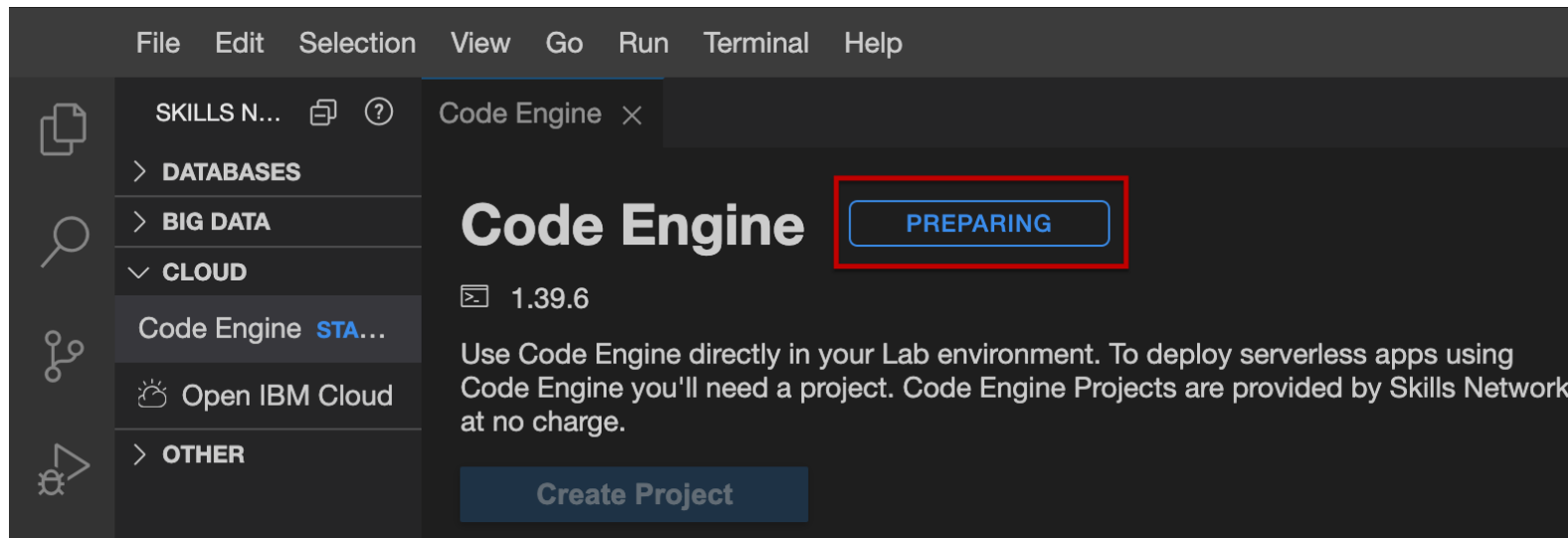
Create Project

Summary Project Information Details

Get started with Code Engine the easy way with your Code Engine Project provided by Skills Network. To create your Code Engine Project, click the button above.

For more details about Code Engine as an IBM Cloud Service, please check out the Details section.

2. The code engine environment takes a while to prepare. You will see the progress status being indicated in the set up panel.



3. Once the code engine set up is complete, you can see that it is active. Click on Code Engine CLI to begin the pre-configured CLI in the terminal below.

The screenshot displays the Skills Network IDE interface. The left sidebar contains a navigation menu with categories: DATABASES, BIG DATA, CLOUD, and OTHER. Under the CLOUD category, 'Code Engine' is listed with a red box around the word 'ACTIVE', and 'Open IBM Cloud' is listed below it. The main panel shows the 'Code Engine' project details. At the top, the title 'Code Engine' is followed by a red box around a green 'READY TO USE' button. Below this, the version '1.39.6' is shown. A paragraph explains that Code Engine can be used directly in the Lab environment for serverless apps. A blue 'Delete Project' button is visible. Below this are tabs for 'Summary', 'Project Information', and 'Details'. The 'Summary' tab is active, showing a message that the project is ready to use. It also provides instructions on where to find more information and a blue 'Code Engine CLI' button, which is also highlighted with a red box.

File Edit Selection View Go Run Terminal Help

SKILLS NETWORK T... Code Engine ×

> DATABASES

> BIG DATA

✓ CLOUD

Code Engine **ACTIVE**

Open IBM Cloud

> OTHER

Code Engine

READY TO USE

1.39.6

Use Code Engine directly in your Lab environment. To deploy serverless apps using Code Engine you'll need a project. Code Engine Projects are provided by Skills Network at no charge.

Delete Project

Summary Project Information Details

Your Skills Network Code Engine Project is now ready to use. You can now create and manage your Serverless Applications.

For important information about your project view the Project Information section. For more details about Code Engine as an IBM Cloud Service, please check out the Details section.

In order to interact with Code Engine please click the following button:

Code Engine CLI

4. You will observe that the pre-configured CLI startup and the home directory is set to the current directory. As a part of the pre-configuration, the project has been set up and Kubeconfig is set up. The details that are shown on the terminal.

```
ibmcloud ce project current
theia@theiadocker-lavanyas:/home/project$ ibmcloud ce project current
Getting the current project context...
OK

Name:      Code Engine - sn-labs-lavanyas
ID:        ee5183a9-4516-4bd1-8f4e-4a8615cafd81
Subdomain: v9oc2xsjxaz
Domain:    us-south.codeengine.appdomain.cloud
Region:    us-south

Kubernetes Config:
Context:    v9oc2xsjxaz
Environment Variable: export KUBECONFIG="/home/theia/.bluemix/plugins/code-engine/Code Engine -
sn-labs-lavanyas-ee5183a9-4516-4bd1-8f4e-4a8615cafd81.yaml"
theia@theiadocker-lavanyas:/home/project$
```

Deploy sentiment analysis on Code Engine as a microservice

1. In the code engine CLI, change to `server/djangoapp/microservices` directory.

```
cd xrwvm-fullstack_developer_capstone/server/djangoapp/microservices
```

You have been provided with `sentiment_analyzer.py` which uses NLTK for sentiment analysis. You are also provided with a `Dockerfile` which you will use to deploy this service in Code Engine and consume it as a microservice. Take a look at these files.

2. Run the following command to docker build the sentiment analyzer app

Please note the code engine instance is transient and is attached to your lab space username.

```
docker build . -t us.icr.io/${SN_ICR_NAMESPACE}/senti_analyzer
```

3. Push the docker image by running the following command.

```
docker push us.icr.io/${SN_ICR_NAMESPACE}/senti_analyzer
```

4. Deploy the senti_analyzer application on code engine.

```
ibmcloud ce application create --name sentianalyzer --image us.icr.io/${SN_ICR_NAMESPACE}/senti_analyzer --registry-secret icr-secret --port 5000
```

5. Connect to the URL that is generated to access the microservices and check if the deployment is successful.

6. If the application deployment verification was successful, attach /analyze/Fantastic services to the URL in the browser to see if it returns **positive**. Take a screenshot of the sentiment along with the URL as shown below and save it as sentiment_analyzer.png OR sentiment_analyzer.jpg.

7. Open djangoapp/.env and replace your code engine deployment url with the deployment URL you obtained above.

It is essential to include the / at the end of the URL. Please ensure that it is copied.

```
sentiment_analyzer_url=your code engine deployment url
```

8. Update djangoapp/restapis.py and add the following function in it to consume the microservice to analyze sentiments.

```
def analyze_review_sentiments(text):
    request_url = sentiment_analyzer_url+"analyze/"+text
    try:
        # Call get method of requests library with URL and parameters
        response = requests.get(request_url)
        return response.json()
    except Exception as err:
        print(f"Unexpected {err=}, {type(err)=}")
        print("Network exception occurred")
```

Create Django views to get dealers

1. Update the get_dealerships view method in djangoapp/views.py with the following code. It will use the get_request you implemented in the restapis.py passing the /fetchDealers endpoint.

```
#Update the `get_dealerships` render list of dealerships all by default, particular state if state is passed
def get_dealerships(request, state="All"):
    if(state == "All"):
        endpoint = "/fetchDealers"
    else:
        endpoint = "/fetchDealers/"+state
    dealerships = get_request(endpoint)
    return JsonResponse({"status":200,"dealers":dealerships})
```

- Configure the route for get_dealerships view method in url.py:

```
path(route='get_dealers', view=views.get_dealerships, name='get_dealers'),
```



```
path(route='get_dealers/<str:state>', view=views.get_dealerships, name='get_dealers_by_state'),
```

2. Create a `get_dealer_details` method which takes the `dealer_id` as a parameter in `views.py` and add a mapping `urls.py`. It will use the `get_request` you implemented in the `restapis.py` passing the `/fetchDealer/<dealer id>` endpoint.

▼ Click here for a sample

Add the following to `views.py`

```
def get_dealer_details(request, dealer_id):
    if(dealer_id):
        endpoint = "/fetchDealer/"+str(dealer_id)
        dealership = get_request(endpoint)
        return JsonResponse({"status":200,"dealer":dealership})
    else:
        return JsonResponse({"status":400,"message":"Bad Request"})
```

Add the following to `urls.py`

```
path(route='dealer/<int:dealer_id>', view=views.get_dealer_details, name='dealer_details'),
```

3. Create `get_dealer_reviews` method which takes the `dealer_id` as a parameter in `views.py` and add a mapping `urls.py`. It will use the `get_request` you implemented in the `restapis.py` passing the `/fetchReviews/dealer/<dealer id>` endpoint. It will also call `analyze_review_sentiments` in `restapis.py` to consume the microservice and determine the sentiment of each of the reviews and set the value in the `review_detail` dictionary which is returned as a `JsonResponse`.

The value of `sentiment` attribute will be determined by sentiment analysis microservice. It could be `positive`, `neutral`, or `negative`.

▼ Click here for sample

Add the following to `views.py`

```
def get_dealer_reviews(request, dealer_id):
    # if dealer id has been provided
    if(dealer_id):
        endpoint = "/fetchReviews/dealer/"+str(dealer_id)
        reviews = get_request(endpoint)
        for review_detail in reviews:
            response = analyze_review_sentiments(review_detail['review'])
            print(response)
            review_detail['sentiment'] = response['sentiment']
        return JsonResponse({"status":200,"reviews":reviews})
    else:
        return JsonResponse({"status":400,"message":"Bad Request"})
```

Add the following to `urls.py`

```
path(route='reviews/dealer/<int:dealer_id>', view=views.get_dealer_reviews, name='dealer_details'),
```

Create a Django view to post a dealer review

By now you have learned how to make various GET calls.

1. Open `restapis.py`, add a `post_review` method which will take a data dictionary in and call the `add_review` in the backend. The dictionary would take all the values required for the dealership review as key-value, pair.

```
def post_review(data_dict):
    request_url = backend_url+"/insert_review"
    try:
        response = requests.post(request_url,json=data_dict)
        print(response.json())
        return response.json()
    except:
        print("Network exception occurred")
```

2. Open `views.py`, create a new `def add_review(request):` method to handle review post request. In the `add_review` view method:
 - First check if user is authenticated because only authenticated users can post reviews for a dealer.
 - Call the `post_request` method with the dictionary
 - Return the result of `post_request` to `add_review` view method. You may print the post response
 - Return a success status and message as JSON
 - Configure the route for `add_review` view in `url.py`.

```
def add_review(request):
    if request.user.is_anonymous == False:
        data = json.loads(request.body)
        try:
            response = post_review(data)
            return JsonResponse({"status":200})
        except:
            return JsonResponse({"status":401,"message":"Error in posting review"})
    else:
        return JsonResponse({"status":403,"message":"Unauthorized"})
```

```
path(route='add_review', view=views.add_review, name='add_review'),
```

3. Import the methods from `restapis.py` for use inside `views.py`.

```
from .restapis import get_request, analyze_review_sentiments, post_review
```

Commit your updated project to GitHub

Commit all updates to the GitHub repository you created so that you can save your work.

If you need to refresh your memory on how to commit and push to GitHub in Theia lab environment, please refer to this lab [Working with git in the Theia lab environment](#)

External References

- [Requests Developer Interface](#)
- [NLTK](#)

Summary

In this lab, you have learned how to create proxy services to call the cloud functions in Django, convert their JSON results into Python objects such as `CarDealer` or `DealerReview`, and return the objects as a `HTTPResonse`.

In the next lab, you will create Django templates to present those objects.

Author(s)

Lavanya T S

Yan Luo

Other Contributor(s)

Upkar Lidder

© IBM Corporation 2024. All rights reserved.