"GeniusGalaxy: AI-Powered Learning for Young Minds"

Welcome to GeniusGalaxy, where the stars align to illuminate the path of young learners. Our platform harnesses the power of artificial intelligence to create an immersive educational experience like no other. By combining cutting-edge technology with tailored curriculum, GeniusGalaxy guides students on a journey of discovery, where every challenge becomes an opportunity to shine. Join us as we embark on a voyage through the cosmos of knowledge, where young minds ignite and brilliance knows no bounds. Welcome to GeniusGalaxy, where the future begins today.

Creating a comprehensive data pipeline that collects data from various sources, stores it in a data warehouse using Spark, processes queries using a large language model (LLM) for converting text to SQL, and integrates machine learning (ML) for further analysis involves several steps. Below is a complete outline and sample code snippets for each component.

### Step 1: Data Collection from Various Sources

* Sources: Google Drive, Azure Blob Storage, AWS S3, CSV, PDF
* Tools: Google Drive API, Azure SDK, AWS SDK (Boto3), PyPDF2 (for PDFs), pandas (for CSV)

### Step 2: Data Ingestion into Spark

* Tools: Apache Spark, PySpark

### Step 3: Data Processing and Storage in Hive

* Tools: Apache Hive, PySpark

### Step 4: Query Processing using LLM

* Tools: OpenAI GPT, SQLAlchemy, PySpark

### Step 5: Data Export to CSV and ML Integration

* Tools: pandas, scikit-learn (Python) or caret (R), Flask API

### Step 6: Visualization

* Tools: matplotlib, seaborn, Flask

### Complete Pipeline Implementation

#### 1. Data Collection

python

import boto3

from azure.storage.blob import BlobServiceClient

import pandas as pd

from pydrive.auth import GoogleAuth

from pydrive.drive import GoogleDrive

import PyPDF2

import io

# Function to download data from AWS S3

def download\_from\_s3(bucket\_name, object\_key, local\_file\_path):

s3 = boto3.client('s3')

s3.download\_file(bucket\_name, object\_key, local\_file\_path)

# Function to download data from Azure Blob Storage

def download\_from\_azure(container\_name, blob\_name, local\_file\_path):

blob\_service\_client = BlobServiceClient.from\_connection\_string("your\_connection\_string")

blob\_client = blob\_service\_client.get\_blob\_client(container=container\_name, blob=blob\_name)

with open(local\_file\_path, "wb") as download\_file:

download\_file.write(blob\_client.download\_blob().readall())

# Function to download data from Google Drive

def download\_from\_drive(file\_id, local\_file\_path):

gauth = GoogleAuth()

gauth.LocalWebserverAuth()

drive = GoogleDrive(gauth)

file = drive.CreateFile({'id': file\_id})

file.GetContentFile(local\_file\_path)

# Function to read data from CSV

def read\_csv(file\_path):

return pd.read\_csv(file\_path)

# Function to read data from PDF

def read\_pdf(file\_path):

with open(file\_path, 'rb') as file:

reader = PyPDF2.PdfFileReader(file)

text = ""

for page in range(reader.numPages):

text += reader.getPage(page).extract\_text()

return text

# Example usage

download\_from\_s3('mybucket', 'data/file.csv', 'local\_file.csv')

download\_from\_azure('mycontainer', 'data/file.csv', 'local\_file.csv')

download\_from\_drive('file\_id', 'local\_file.csv')

csv\_data = read\_csv('local\_file.csv')

pdf\_data = read\_pdf('local\_file.pdf')

#### 2. Data Ingestion into Spark

python

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName("DataPipeline") \

.config("spark.sql.warehouse.dir", "hdfs://localhost:9000/user/hive/warehouse") \

.enableHiveSupport() \

.getOrCreate()

# Ingest CSV data into Spark DataFrame

df = spark.read.csv('local\_file.csv', header=True, inferSchema=True)

df.show()

# Save DataFrame to Hive

df.write.mode("overwrite").saveAsTable("default.my\_table")

#### 3. Query Processing using LLM

python

from transformers import pipeline

# Load a pre-trained GPT model for text-to-SQL conversion

llm = pipeline('text-generation', model='gpt-3.5-turbo')

def text\_to\_sql(query\_text):

response = llm(f"Convert this text to SQL: {query\_text}")

sql\_query = response[0]['generated\_text']

return sql\_query

# Example usage

query\_text = "Get all records where age is greater than 30"

sql\_query = text\_to\_sql(query\_text)

print(sql\_query)

#### 4. Data Export to CSV and ML Integration

python

import sqlite3

# Execute SQL query in Spark

result\_df = spark.sql(sql\_query)

# Save result to CSV

result\_df.toPandas().to\_csv('result.csv', index=False)

# Example ML integration (Python)

def run\_ml\_model(csv\_path):

data = pd.read\_csv(csv\_path)

# Assuming the ML model is a simple sklearn model

from sklearn.linear\_model import LinearRegression

model = LinearRegression()

X = data[['feature1', 'feature2']] # Adjust according to your data

y = data['target']

model.fit(X, y)

predictions = model.predict(X)

return predictions

predictions = run\_ml\_model('result.csv')

print(predictions)

# API to get ML output (Flask)

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

@app.route('/get\_ml\_output', methods=['POST'])

def get\_ml\_output():

file\_path = request.json['file\_path']

predictions = run\_ml\_model(file\_path)

return jsonify(predictions.tolist())

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

#### 5. Visualization

python

import matplotlib.pyplot as plt

import seaborn as sns

def plot\_predictions(predictions):

sns.histplot(predictions, kde=True)

plt.title('Predictions Distribution')

plt.xlabel('Prediction')

plt.ylabel('Frequency')

plt.savefig('predictions\_plot.png')

plt.show()

# Example usage

plot\_predictions(predictions)

### Pipeline Summary

1. Data Collection: Use APIs and SDKs to collect data from various sources.
2. Data Ingestion: Ingest data into Spark DataFrames and save to Hive.
3. Query Processing: Use LLM to convert natural language queries to SQL and execute in Spark.
4. Data Export: Export query results to CSV and run ML models.
5. Visualization: Plot the ML results using visualization libraries.
6. User Authentication and Session Management:
   * Implement user authentication to allow users to log in.
   * Use session management to maintain user sessions.
   * Ensure that each user's actions are associated with their session.
7. Database Schema:
   * Modify your MongoDB schema to include fields for:
     + User details (username, email, password, etc.).
     + Input file details (file name, path, user ID).
     + Function details (function name, input and output file names, user ID).
     + Model details (model name, function ID, user ID, etc.).
8. File Storage:
   * Store uploaded input files in a directory structure that includes the user's username and function name combined.
   * Ensure that each user can only access their own files.
9. Function Execution:
   * When a user selects a function to execute, retrieve the input file path associated with their session and the selected function.
   * Execute the function on the input file.
   * Save the output to a file using the user's username and function name combined.
10. Activation Functions:

* Write scripts or functions in Python using PyTorch or TensorFlow to perform various activation functions, all ML functions etc with API take input as file name userId\_fileName( based on Pipeline Function which is used previously).
* Expose these functions as APIs using packages like Flask or FastAPI.
* Integrate these APIs into your Node.js or Springe boot backend using HTTP requests.

1. Apply Function Endpoint:

* Create an endpoint in your Express.js app to apply selected activation function to the uploaded file.
* Retrieve the file from the server based on the user's session ID.
* Call the corresponding Python API for the selected activation function and pass the file data.
* Save the output file with a unique name( basedon userId\_function\_used) and store its details in the database.

1. Database Operations:
   * When a user creates a new model, store the details in the database, including the user ID, function ID, and any other relevant information.
   * Ensure that each model is associated with the correct user.
2. Access Control:
   * Implement access control to ensure that users can only access their own data.
   * Use user IDs and session tokens to enforce access control rules.
3. Error Handling and Validation:
   * Validate user input to prevent errors and security vulnerabilities.
   * Handle errors gracefully to provide a smooth user experience.
4. Testing and Deployment:
   * Test the application thoroughly to ensure that it works as expected, especially in a multi-user environment.
   * Deploy the application to a server or cloud platform for production use.

By following these steps, you can create a tool using the full stack that allows users to upload data, select ML/DL functions using icons, and process the data to get outputs similar to MATLAB.

**Here's how this setup can benefit your project:**

1. Django for AI Functions:
   * Django's integration with Python makes it an excellent choice for implementing AI functions using libraries like TensorFlow or PyTorch.
   * You can create RESTful APIs in Django to expose these AI functions, allowing other parts of your application to interact with them.
2. Node.js or Spring Boot for Web Frontend:
   * Node.js or Spring Boot can be used to develop the web frontend of your application, providing a responsive and interactive user interface.
   * These platforms excel in handling client-side interactions, making them suitable for building dynamic web applications.
3. Node.js or Spring Boot for Backend Admin Panel:
   * You can also use Node.js or Spring Boot to develop the backend admin panel application, providing functionalities for managing users, content, and other administrative tasks.
   * These platforms offer robust backend capabilities and can integrate seamlessly with your Django API.
4. Scalability and Performance:
   * Separating the AI functions from the frontend and backend applications allows for better scalability and performance optimization.
   * You can scale each component independently based on the specific requirements and workload of your application.
5. Team Expertise and Flexibility:
   * Leveraging Django for AI functions may be beneficial if your team is proficient in Python and prefers Django's development workflow.
   * Using Node.js or Spring Boot for the frontend and backend applications provides flexibility and allows you to choose the technology stack that best fits your team's skills and preferences.

Overall, this architecture enables you to build a robust and scalable educational platform that leverages the strengths of Django, Node.js, or Spring Boot, depending on the specific requirements of each component. It also allows for easier maintenance and future expansion of your application.

**To create a full-stack application using Angular for the front-end, Spring Boot for the backend, and Django for the Python API, we will follow these steps:**

1. Set up the front-end with Angular to allow users to select data sources and input required connection details.
2. Set up the backend with Spring Boot to handle requests from the front-end and interact with the Django API.
3. Set up the Python API with Django to handle data collection from various sources.

### Step 1: Set Up the Angular Front-End

#### 1.1 Create Angular Components

1. Install Angular CLI (if not already installed):

bash

* npm install -g @angular/cli
* Create a new Angular project:

bash

* ng new data-collector

cd data-collector

* Generate components for each data source and a form to collect user input:

bash

* ng generate component data-source-selector

ng generate component data-source-form

* Install necessary Angular Material and icons:

bash

1. ng add @angular/material

#### 1.2 Data Source Selector Component

data-source-selector.component.html:

html

<div class="data-source-icons">

<mat-icon (click)="selectDataSource('s3')">cloud</mat-icon>

<mat-icon (click)="selectDataSource('azure')">cloud\_queue</mat-icon>

<mat-icon (click)="selectDataSource('gdrive')">folder</mat-icon>

<mat-icon (click)="selectDataSource('csv')">description</mat-icon>

<mat-icon (click)="selectDataSource('pdf')">picture\_as\_pdf</mat-icon>

</div>

data-source-selector.component.ts:

typescript

import { Component, EventEmitter, Output } from '@angular/core';

@Component({

selector: 'app-data-source-selector',

templateUrl: './data-source-selector.component.html',

styleUrls: ['./data-source-selector.component.css']

})

export class DataSourceSelectorComponent {

@Output() dataSourceSelected = new EventEmitter<string>();

selectDataSource(source: string) {

this.dataSourceSelected.emit(source);

}

}

#### 1.3 Data Source Form Component

data-source-form.component.html:

html

<div \*ngIf="selectedSource">

<form [formGroup]="dataSourceForm" (ngSubmit)="onSubmit()">

<div \*ngIf="selectedSource === 's3'">

<mat-form-field>

<mat-label>AWS Access Key</mat-label>

<input matInput formControlName="awsAccessKey" required>

</mat-form-field>

<mat-form-field>

<mat-label>AWS Secret Key</mat-label>

<input matInput formControlName="awsSecretKey" required>

</mat-form-field>

<mat-form-field>

<mat-label>Bucket Name</mat-label>

<input matInput formControlName="bucketName" required>

</mat-form-field>

<mat-form-field>

<mat-label>Object Key</mat-label>

<input matInput formControlName="objectKey" required>

</mat-form-field>

</div>

<!-- Add similar form fields for Azure, Google Drive, CSV, and PDF -->

<button mat-raised-button type="submit">Submit</button>

</form>

</div>

data-source-form.component.ts:

typescript

import { Component, Input, OnChanges } from '@angular/core';

import { FormBuilder, FormGroup } from '@angular/forms';

import { HttpClient } from '@angular/common/http';

@Component({

selector: 'app-data-source-form',

templateUrl: './data-source-form.component.html',

styleUrls: ['./data-source-form.component.css']

})

export class DataSourceFormComponent implements OnChanges {

@Input() selectedSource: string;

dataSourceForm: FormGroup;

constructor(private fb: FormBuilder, private http: HttpClient) {

this.dataSourceForm = this.fb.group({

awsAccessKey: [''],

awsSecretKey: [''],

bucketName: [''],

objectKey: ['']

// Add form controls for other data sources as well

});

}

ngOnChanges() {

if (this.selectedSource) {

this.dataSourceForm.reset();

}

}

onSubmit() {

const formData = this.dataSourceForm.value;

this.http.post(`/api/data-collect/${this.selectedSource}`, formData).subscribe(response => {

console.log('Data collected:', response);

});

}

}

#### 1.4 App Component

app.component.html:

html

<app-data-source-selector (dataSourceSelected)="onDataSourceSelected($event)"></app-data-source-selector>

<app-data-source-form [selectedSource]="selectedSource"></app-data-source-form>

app.component.ts:

typescript

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

selectedSource: string;

onDataSourceSelected(source: string) {

this.selectedSource = source;

}

}

### Step 2: Set Up the Backend with Spring Boot

1. Create a new Spring Boot project using [Spring Initializr](https://start.spring.io/):
   * Dependencies: Spring Web, Spring Boot DevTools, Spring Data JPA, MySQL Driver (or H2 Database for simplicity)
2. Create a controller to handle API requests from the Angular frontend.

DataCollectorController.java:

java

package com.example.datacollector;

import org.springframework.web.bind.annotation.\*;

import org.springframework.beans.factory.annotation.Autowired;

@RestController

@RequestMapping("/api/data-collect")

public class DataCollectorController {

@Autowired

private PythonService pythonService;

@PostMapping("/{source}")

public ResponseEntity<String> collectData(@PathVariable String source, @RequestBody Map<String, String> params) {

String result = pythonService.collectDataFromSource(source, params);

return ResponseEntity.ok(result);

}

}

PythonService.java:

java

package com.example.datacollector;

import org.springframework.stereotype.Service;

import java.util.Map;

import java.io.BufferedReader;

import java.io.InputStreamReader;

@Service

public class PythonService {

public String collectDataFromSource(String source, Map<String, String> params) {

try {

ProcessBuilder processBuilder = new ProcessBuilder("python3", "path/to/your/script.py", source, params.toString());

Process process = processBuilder.start();

BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));

StringBuilder output = new StringBuilder();

String line;

while ((line = reader.readLine()) != null) {

output.append(line).append("\n");

}

return output.toString();

} catch (Exception e) {

e.printStackTrace();

return "Error collecting data";

}

}

}

### Step 3: Set Up the Python API with Django

1. Create a new Django project and app:

bash

1. django-admin startproject datacollector

cd datacollector

django-admin startapp api

1. Create views to handle data collection from various sources.

api/views.py:

python

from django.http import JsonResponse

from django.views.decorators.csrf import csrf\_exempt

import boto3

from azure.storage.blob import BlobServiceClient

from pydrive.auth import GoogleAuth

from pydrive.drive import GoogleDrive

import pandas as pd

import PyPDF2

import io

import json

@csrf\_exempt

def collect\_data(request, source):

if request.method == 'POST':

params = json.loads(request.body)

if source == 's3':

return collect\_from\_s3(params)

elif source == 'azure':

return collect\_from\_azure(params)

elif source == 'gdrive':

return collect\_from\_gdrive(params)

elif source == 'csv':

return collect\_from\_csv(params)

elif source == 'pdf':

return collect\_from\_pdf(params)

return JsonResponse({'status': 'Invalid request'}, status=400)

def collect\_from\_s3(params):

s3 = boto3.client('s3', aws\_access\_key\_id=params['awsAccessKey'], aws\_secret\_access\_key=params['awsSecretKey'])

s3.download\_file(params['bucketName'], params['objectKey'], '/tmp/data.csv')

data = pd.read\_csv('/tmp/data.csv')

return JsonResponse({'status': 'success', 'data': data.to\_dict()})

def collect\_from\_azure(params):

blob\_service\_client = BlobServiceClient.from\_connection\_string(params['connectionString'])

blob\_client = blob\_service\_client.get\_blob\_client(container=params['containerName'], blob=params['blobName'])

with open('/tmp/data.csv', "wb") as download\_file:

download\_file.write(blob\_client.download\_blob().readall())

data = pd.read\_csv('/tmp/data.csv')

return JsonResponse({'status': 'success', 'data': data.to\_dict()})

def collect\_from\_gdrive(params):

gauth = GoogleAuth()

gauth.LocalWebserverAuth()

drive = GoogleDrive(gauth)

file = drive.CreateFile({'id': params['fileId']})

file.GetContentFile('/tmp/data.csv')

data = pd.read\_csv('/tmp/data.csv')

return JsonResponse({'status': 'success', 'data': data.to\_dict()})

def collect\_from\_csv(params):

data = pd.read\_csv(params['filePath'])

return JsonResponse({'status': 'success', 'data': data.to\_dict()})

def collect\_from\_pdf(params):

with open(params['filePath'], 'rb') as file:

reader = PyPDF2.PdfFileReader(file)

text = ""

for page in range(reader.numPages):

text += reader.getPage(page).extract\_text()

return JsonResponse({'status': 'success', 'data': text})

1. Configure URLs to route requests to the view.

api/urls.py:

python

from django.urls import path

from .views import collect\_data

urlpatterns = [

path('collect/<str:source>/', collect\_data),

]

datacollector/urls.py:

python

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('api/', include('api.urls')),

]

### Integration

1. Run the Django server:

bash

* python manage.py runserver
* Run the Spring Boot application.
* Run the Angular application:

bash

1. ng serve

### Final Notes

* Security: Add proper authentication and authorization to the backend APIs.
* Error Handling: Implement robust error handling in all parts of the system.
* Optimization: Optimize the data collection process, especially for large datasets.

**Spring Boot serves as the backend server** that acts as an intermediary between the Angular front-end and the Django API. Its primary role is to handle client requests, coordinate the interaction with the Python-based Django service, and ensure smooth data flow and processing. Here’s a detailed explanation of what Spring Boot is doing in this setup:

### Responsibilities of Spring Boot

1. Handle Client Requests: Spring Boot receives API requests from the Angular front-end. It processes these requests, gathers necessary data, and then forwards them to the appropriate service (in this case, the Django API).
2. Coordinate with Django API: Spring Boot forwards the data collection requests to the Django API, which performs the actual data collection from various sources. After Django completes its task, Spring Boot receives the response and processes it accordingly.
3. Data Processing and Formatting: If needed, Spring Boot can process and format the data received from the Django API before sending it back to the Angular front-end.
4. Error Handling and Logging: Spring Boot provides a centralized mechanism for error handling and logging, which helps in maintaining a robust and traceable application.
5. Security and Authentication: Spring Boot can be used to implement security features such as authentication and authorization, ensuring that only authorized users can access the data collection functionality.
6. Scalability and Management: Spring Boot helps in scaling the backend services as per the load and managing configurations centrally, making the application more maintainable and scalable.

### Example Implementation in Spring Boot

1. Set up a Spring Boot project using [Spring Initializr](https://start.spring.io/).
2. Add Dependencies: Ensure you have dependencies for Spring Web and Spring Boot DevTools.
3. Create a Controller to Handle Requests:

DataCollectorController.java:

java

package com.example.datacollector;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.Map;

@RestController

@RequestMapping("/api/data-collect")

public class DataCollectorController {

@Autowired

private DjangoApiService djangoApiService;

@PostMapping("/{source}")

public ResponseEntity<String> collectData(@PathVariable String source, @RequestBody Map<String, String> params) {

String result = djangoApiService.collectDataFromSource(source, params);

return ResponseEntity.ok(result);

}

}

1. Create a Service to Interact with Django API:

DjangoApiService.java:

java

package com.example.datacollector;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

import java.util.Map;

@Service

public class DjangoApiService {

private final RestTemplate restTemplate;

public DjangoApiService() {

this.restTemplate = new RestTemplate();

}

public String collectDataFromSource(String source, Map<String, String> params) {

String url = "http://localhost:8000/api/collect/" + source + "/";

return restTemplate.postForObject(url, params, String.class);

}

}

### Angular Front-End Interaction

1. Angular Service to Call Spring Boot API:

data-collector.service.ts:

typescript

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class DataCollectorService {

private apiUrl = '/api/data-collect';

constructor(private http: HttpClient) { }

collectData(source: string, params: any): Observable<any> {

return this.http.post(`${this.apiUrl}/${source}`, params);

}

}

1. Integrate Angular Components with the Service:

data-source-form.component.ts:

typescript

import { Component, Input, OnChanges } from '@angular/core';

import { FormBuilder, FormGroup } from '@angular/forms';

import { DataCollectorService } from '../services/data-collector.service';

@Component({

selector: 'app-data-source-form',

templateUrl: './data-source-form.component.html',

styleUrls: ['./data-source-form.component.css']

})

export class DataSourceFormComponent implements OnChanges {

@Input() selectedSource: string;

dataSourceForm: FormGroup;

constructor(

private fb: FormBuilder,

private dataCollectorService: DataCollectorService

) {

this.dataSourceForm = this.fb.group({

awsAccessKey: [''],

awsSecretKey: [''],

bucketName: [''],

objectKey: ['']

// Add form controls for other data sources as well

});

}

ngOnChanges() {

if (this.selectedSource) {

this.dataSourceForm.reset();

}

}

onSubmit() {

const formData = this.dataSourceForm.value;

this.dataCollectorService.collectData(this.selectedSource, formData).subscribe(response => {

console.log('Data collected:', response);

});

}

}

### Summary

* Angular Front-End: Provides a user interface for selecting data sources and inputting connection details. It sends user requests to the Spring Boot backend.
* Spring Boot Backend: Acts as an intermediary that processes requests from Angular, forwards them to the Django API, handles responses, and implements additional backend logic if necessary.
* Django API: Handles the actual data collection from various sources (Google Drive, Azure Blob Storage, AWS S3, CSV, PDF) and returns the collected data to Spring Boot.

This architecture leverages the strengths of each technology to create a robust, scalable, and maintainable full-stack application.

+-----------------------+ +-------------------+ +-----------------------+

| | | | | |

| Angular Front-End | | Spring Boot | | Django API |

| (User Interface) | | Backend | | (Data Collection) |

| | | | | |

| +----------------+ | | +------------+ | | +----------------+ |

| | Data Source | | | | | | | | Handle | |

| | Selector +----------> | | +--------> | | Data | |

| +----------------+ | | | | | | | Collection | |

| | | | | | | +----------------+ |

| +----------------+ | | | | | | |

| | Data Source | | | | Django API | | | +----------------+ |

| | Form +----------> | | Service +--------> | | Collect from | |

| +----------------+ | | | | | | | AWS S3 | |

| | | | | | | +----------------+ |

| +----------------+ | | | | | | |

| | HTTP Client | | | | | | | +----------------+ |

| +----------------+ | | +------------+ | | | Collect from | |

| | | | | | Azure Blob | |

| | +-------------------+ | +----------------+ |

| | | |

| | +-------------------+ | +----------------+ |

| | | | | | Collect from | |

| | | | | | Google Drive | |

| | | | | +----------------+ |

| | | | | |

| | +-------------------+ | +----------------+ |

| | | | Collect from | |

| | | | CSV | |

| | | +----------------+ |

| | | |

| | | +----------------+ |

| | | | Collect from | |

| | | | PDF | |

| | | +----------------+ |

|  |  |  |
| --- | --- | --- |

### Explanation

#### Angular Front-End:

* Data Source Selector Component: Allows the user to select the data source (AWS S3, Azure Blob Storage, Google Drive, CSV, PDF).
* Data Source Form Component: Displays a form for the user to input connection details for the selected data source.
* HTTP Client: Sends the user's input and selected data source information to the Spring Boot backend.

#### Spring Boot Backend:

* Controller: Receives API requests from the Angular front-end.
* Django API Service: Forwards the request to the Django API, passing along the necessary connection details.
* Response Handling: Processes the response from the Django API and forwards the relevant data back to the Angular front-end.

#### Django API:

* Data Collection Handlers: Individual handlers for each data source (AWS S3, Azure Blob Storage, Google Drive, CSV, PDF). These handlers collect data using the provided connection details and return the collected data back to the Spring Boot backend.

### Interaction Flow:

1. User Interaction:
   * The user selects a data source using the Angular front-end.
   * The user inputs connection details and submits the form.
2. Request Handling:
   * Angular sends the request to the Spring Boot backend.
   * Spring Boot receives the request and calls the Django API service with the necessary parameters.
3. Data Collection:
   * The Django API handles the data collection based on the specified data source and connection details.
   * Django returns the collected data to the Spring Boot backend.
4. Response Handling:
   * Spring Boot processes the response from Django and sends the collected data back to the Angular front-end.
5. Display Data:
   * Angular displays the collected data to the user.

This architecture separates the concerns of front-end user interaction, backend request handling, and data collection, providing a modular and maintainable full-stack application.