Technical Specifications Document for Weather App

Revision History

Approver Role	Name	Signature
Integration Developer	Diway Sanu diwaygrover0019@gmail.com https://github.com/diwaygrover0019	Version 1.0

Contents

Introduction

Weather App exposes two REST endpoints, fetching data from downstream SOAP webservice (http://www.webservicex.com/globalweather.asmx?WSDL). Weather App is developed using Spring Boot, along with Web Flux.

Assumptions

- Provided downstream service URL http://www.webservicex.com/globalweather.asmx?WSDL is not available. Instead we are using the NodeJS/Docker version of webservice.
- Extracted the provided zip, and ran the NodeJS service in my local machine.
- Provided NodeJS service response are not standard SOAP based response, so downstream data should always be per the mocks i.e.
    - getCities response is encapsulated content with **<![CDATA[**content**]]>**
    - getWeather response is encapsulated content with **<![CDATA[<![CDATA[**content**]]>]]>**
- If content is not as per the mock, then exception will be raised and relevant message will be shown.
- The code should be deployed using JDK 8 or JDK 11.
- Apache CXF plugin has been used; alternate could have been Jaxb2 plugin.

Environment Details

• spring-boot-starter-parent	:	2.2.5.RELEASE
• JDK	:	8/11
• Swagger	:	2.0
• Apache CXF plugin	:	3.3.5
• Jacoco	:	0.8.1

Prerequisites

- As external SOAP webservice is not available, NodeJS service has been deployed on local machine (http://localhost:8080/GlobalWeather), below are the steps:
    i. Extract the zip
    ii. Run npm install
    iii. Run npm start
- If you want to use different properties for different environments, e.g. for prod one can set `spring.profiles.active=prod` in application.properties file.

REST API Endpoints

Weather App exposes below two REST endpoints:

GET /api/getCities

Description: Get all the cities for the provided country name in query parameters. If the query parameter is not passed, API will give 400 (Bad Request) error.

Query Parameter:

Parameter	Type	Required	Example and URI
country	String	Yes	country=Australia

Example URI: http://localhost:8081/api/getCities?country=Australia

Response Headers:

Content-Type: **application/json**

Response Body:

Schema:

```
{
  "cities": [
    {
      "city": "string",
      "country": "string"
    }
  ]
}
```

GET /api/getWeather

Description: Get the weather data for the provided city name and country name in query parameters. If the query parameter is not passed, API will give 400 (Bad Request) error.

Query Parameter:

Parameter	Type	Required	Example and URI
city	String	Yes	city=Sydney
country	String	Yes	country=Australia

Example URI: http://localhost:8081/api/getWeather?city=Sydney&country=Australia

Response Headers:

Content-Type: **application/json**

Response Body:

Schema:

```
{
  "dewPoint": "string",
  "location": "string",
  "relativeHumidity": "string",
  "skyConditions": "string",
  "status": "string",
  "temperature": "string",
  "time": "string",
  "visibility": "string",
  "wind": "string"
}
```

The **Swagger** is attached below for the above endpoints (if unable to open directly, please drag and drop):

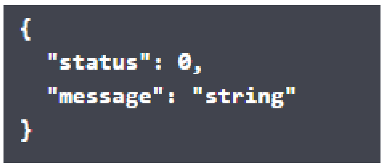
swagger.yaml

Error Handling

Weather App handles all major Exceptions including Runtime Exception's. We have a customized **ErrorResponse** object which we return to the consumers of the APIs.

Error Response Body:

Schema:



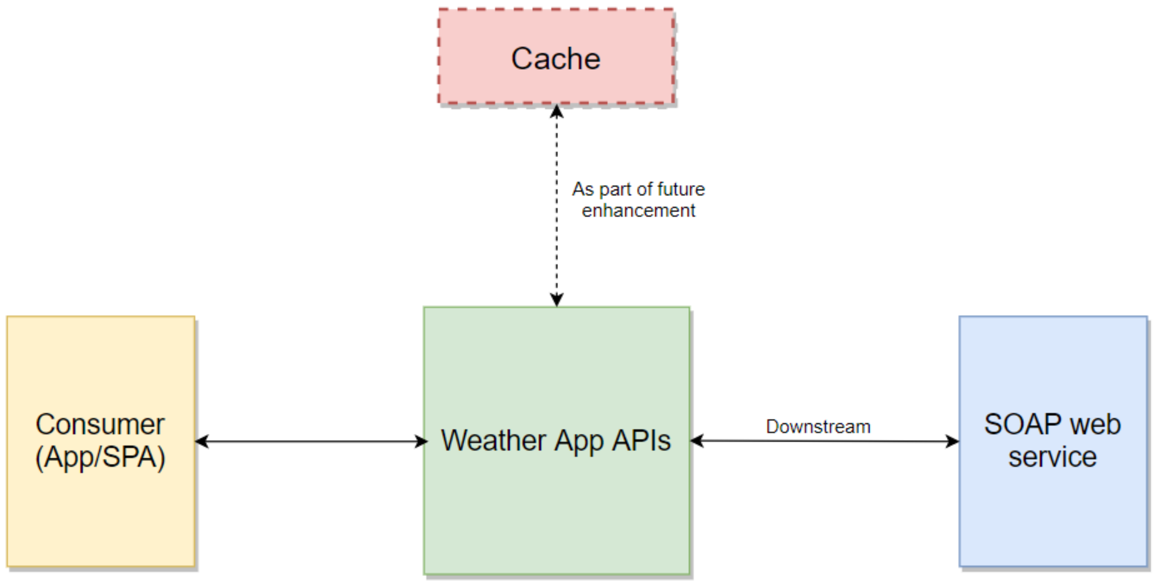
```
{
  "status": 0,
  "message": "string"
}
```

Implementation flow handles below error:

- **/api/getCities** and ***/api/getWeather***
  - 400 (Bad request) : "if user enters null or invalid query parameters"
  - 500 (Internal Server Error) : "if any unwanted exception occurs"
    - This includes any downstream error handling
    - Interrupted/Execution exception due to CompletableFuture
    - Marshalling Failure Exception

API Architectural Flow and Capabilities

Architecture flow



Project Capabilities

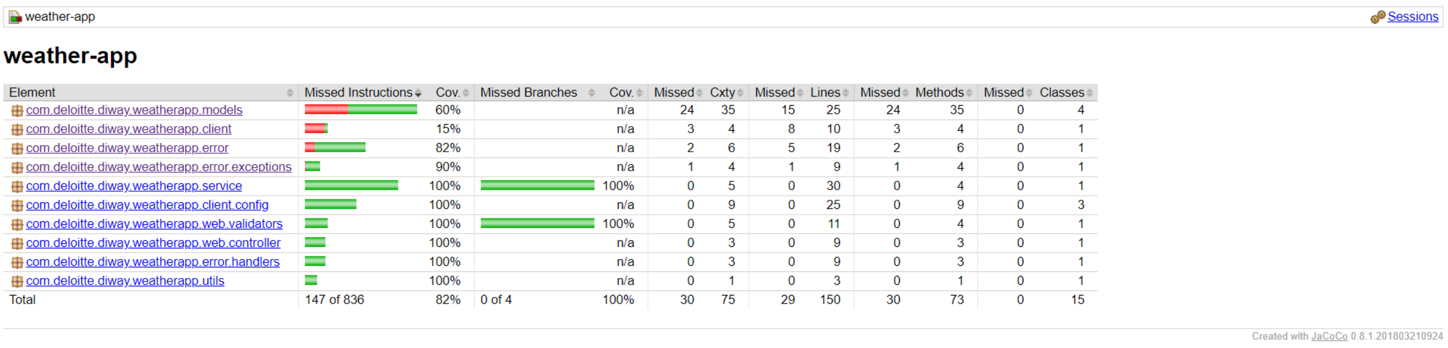
The following additional features has been added to enhance application capabilities:

Jacoco

- Jacoco plugin has been added to check code coverage.
- To run Jacoco plugin, run the below command:

```
mvn clean jacoco:prepare-agent install -Dmaven.test.failure.ignore=false
```

- The coverage report will be available at: `/target/site/jacoco/index.html`
- Please find below coverage report snapshot:



weather-app

Sessions

weather-app

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.deloitte.diway.weatherapp.models		60%		n/a	24	35	15	25	24	35	0	4
com.deloitte.diway.weatherapp.client		15%		n/a	3	4	8	10	3	4	0	1
com.deloitte.diway.weatherapp.error		82%		n/a	2	6	5	19	2	6	0	1
com.deloitte.diway.weatherapp.error.exceptions		90%		n/a	1	4	1	9	1	4	0	1
com.deloitte.diway.weatherapp.service		100%		100%	0	5	0	30	0	4	0	1
com.deloitte.diway.weatherapp.client.config		100%		n/a	0	9	0	25	0	9	0	3
com.deloitte.diway.weatherapp.web.validators		100%		100%	0	5	0	11	0	4	0	1
com.deloitte.diway.weatherapp.web.controller		100%		n/a	0	3	0	9	0	3	0	1
com.deloitte.diway.weatherapp.error.handlers		100%		n/a	0	3	0	9	0	3	0	1
com.deloitte.diway.weatherapp.utils		100%		n/a	0	1	0	3	0	1	0	1
Total	147 of 836	82%	0 of 4	100%	30	75	29	150	30	73	0	15

Created with JaCoCo 0.8.1.201803210924

CompletableFuture

- Any time-consuming task should be done asynchronously.
- CompletableFuture will run the task asynchronously, as we have supplied our custom executor to define the ThreadPool.

Reactive programming

- Reactive programming requires achieving asynchronous, non-blocking functionality.
- We achieved partial reactive programming, as after we have got the downstream response, we are applying some transformation operations, which are kind of blocking practice.
- Post the transformation, we are returning model objects of type **Mono** for the consumers.

Unit Test Case

- SpringBootTest has been used to cover the scope of integration tests.
- Junit and Mockito has been used to write unit test cases along with integration tests.

Enhancement

- **Authorization** can be set on the APIs, which can be based on JWT, Oauth 2.0 or any other standard authorization architecture.
- **Cache** mechanism to save previous responses and use that when possible.
- Implement end-to-end reactive approach.
- Hystrix can be used to implement Circuit Breaker Pattern by failing fast in case downstream service goes down.

- **E2E testing** can be implemented instead of mock responses (currently in our code base).
- Implement better representation of the response data, e.g. setting extra attributes like correlation-id, timestamp, etc.

Challenges

- Response from SOAP webservice is not standard xml response.
- Parsing the downstream response containing CDATA.
- Creating a common modify response method for both APIs as could see one of the API has CDATA embedded twice in response.
- Figuring out which plugin to be used, like Apache CXF or Jaxb2 to generate Java sources from wsdl.

References

- https://www.devglan.com/spring-boot/spring-boot-soap-client
- https://spring.io/guides/gs/consuming-web-service/
- https://stackoverflow.com/questions/23093897/sending-xmldata-in-soap-request-using-spring-ws