AI ASSISTED CODING

END LAB EXAM

NAME : G.RISHIKESH

BATCH : 03

HT.NO : 2403A52046

SET : 10

A)) REFACTORING LEGACY CITY SYSTEMS

Q1: WRAP OLD SOAP SERVICE WITH MODERN REST FACADE.

• **TASK 1**: USE AI TO GENERATE WRAPPER ENDPOINTS AND TRANSFORM PAYLOADS.

PROMPT :

"CREATE REST ENDPOINTS THAT INTERNALLY CALL AN OLD SOAP SERVICE.

* CONVERT INCOMING REST JSON REQUESTS INTO SOAP XML FORMAT.

* INVOKE THE SOAP WSDL OPERATIONS.

* PARSE SOAP XML RESPONSE AND RETURN JSON.

PROVIDE CODE FOR CONTROLLERS, SERIALIZERS, AND TRANSFORMATION LOGIC."

CODE :

```
ATCODING > JS JS.js > ...
1   const express = require("express");
2   const soap = require("soap");
3
4   const app = express();
5   app.use(express.json());
6
7   const WSDL = "https://example.com/service?wsdl";   // put your WSDL URL
8
9   app.post("/api/:op", async (req, res) => {
10    try {
11      const client = await soap.createClientAsync(WSDL);
12      const op = req.params.op;                  // operation name
13      const args = req.body;                     // JSON input
14
15      if (!client[op + "Async"])
16        return res.status(400).json({ error: "SOAP operation not found" });
17
18      const [response] = await client[op + "Async"](args);
19
20      res.json(response);                         // return SOAP response as JSON
21    } catch (err) {
22      res.status(500).json({ error: err.message });
23    }
24  });
25
26  app.listen(3000, () => console.log("REST → SOAP running"));
27
```

OUTPUT :

1) Calling Get User

POST ⟶ http://localhost:3000/api/GetUser

Body:

{ "userId": 5 }

Output:

{

"GetUserResult": {

"userId": 5,

"name": "John Doe",

"email": "john@example.com",

"status": "Active"

}

}

2) Calling Create User

POST:

{

"name": "Alice",

"email": "alice@test.com"

}

Output:

{

"CreateUserResult": {

"status": "Success",

"userId": 101 }

}

OBSERVATION :

1.AI scans the SOAP WSDL to detect operations, parameters, and namespaces.

2.Identifies which REST endpoints map to each SOAP action.

3.Auto-generates JSON ⟶ SOAP XML request transformers.

4.Builds XML ⟶ JSON response parsers while flattening nested SOAP nodes.

5.Ensures SOAP envelope, headers, and types remain compliant.

• TASK 2: ADD TESTS AND DOCUMENTATION

PROMPT :

"Create unit tests for REST-to-SOAP wrapper.

Include:

* MOCK SOAP RESPONSES

* VALIDATION OF XML GENERATION

* RESPONSE JSON FORMATTING

ALSO GENERATE DOCUMENTATION DESCRIBING EACH ENDPOINT, SAMPLE REQUESTS, ERRORS, AND TRANSFORMATION FLOW."

CODE :

```
const express = require("express");
const app = express();
app.use(express.json());
function mockSoap(operation, args) {
  const responses = {
    GetUser: {
      GetUserResult: {
        userId: args.userId,
        name: "John Doe",
        email: "john@example.com",
        status: "Active"
      }
    },
    CreateUser: {
      CreateUserResult: {
        status: "Success",
        userId: 101
      }
    }
  };
  return responses[operation] || { error: "SOAP operation not found" };
}
app.post("/api/:op", (req, res) => {
  const op = req.params.op;
  const args = req.body;

  const result = mockSoap(op, args);

  if (result.error) return res.status(400).json(result);

  res.json(result);
});
module.exports = app;
if (require.main === module) {
  app.listen(3000, () => console.log("REST → SOAP running on port 3000"));
}
if (process.env.JEST_WORKER_ID !== undefined) {
  const request = require("supertest");
```

```
describe("REST → SOAP API TESTS", () => {

  test("GetUser should return correct user", async () => {
    const res = await request(app)
      .post("/api/GetUser")
      .send({ userId: 5 });

    expect(res.statusCode).toBe(200);
    expect(res.body.GetUserResult.userId).toBe(5);
  });

  test("CreateUser should return success", async () => {
    const res = await request(app)
      .post("/api/CreateUser")
      .send({ name: "Alice", email: "alice@test.com" });

    expect(res.statusCode).toBe(200);
    expect(res.body.CreateUserResult.status).toBe("Success");
  });

  test("Unknown operation should return error", async () => {
    const res = await request(app)
      .post("/api/NoSuchOp")
      .send({});

    expect(res.statusCode).toBe(400);
    expect(res.body.error).toBe("SOAP operation not found");
  });
});
}
```

OUTPUT :

OUTPUT FOR /API/GET USER


```
{
 "GetUserResult": {
  "userId": 5,
  "name": "John Doe",
  "email": "john@example.com",
  "status": "Active"
 }
}
```


OUTPUT FOR /API / CREATE USER


```
{
 "CreateUserResult": {
  "status": "Success",
  "userId": 101
 }
}
```


SAMPLE ERROR OUTPUT


```
{
 "error": "SOAP operation not found"
}
```

OBSERVATION :

1,AI SETS UP JEST MOCKS FOR SOAP CLIENT CALLS.

2.VALIDATES THAT EACH REST ENDPOINT TRIGGERS THE RIGHT SOAP ACTION.

3.CONFIRMS JSON TRANSFORMATION CORRECTNESS AND ERROR HANDLING.

4.GENERATES ENDPOINT DOCUMENTATION WITH INPUTS/OUTPUTS.

5.ADDS EXAMPLE PAYLOADS FOR QUICK DEVELOPER REFERENCE**.**

B)) CONSOLIDATE DIFFERENT ALERTING FORMATS INTO UNIFIED SCHEMA.

• **TASK 1**: USE AI TO MAP AND NORMALIZE.

PROMPT :

"ANALYZE MULTIPLE ALERT FORMATS (JSON, XML, CSV) AND DERIVE A UNIFIED SCHEMA.

IDENTIFY COMMON FIELDS SUCH AS:

* ALERT ID

* SOURCE

* TIMESTAMP

* SEVERITY

* DESCRIPTION

GENERATE MAPPING RULES FOR EACH SOURCE FORMAT."

CODE :

```
const xml2js = require("xml2js");
const csv = require("csv-parse/sync");
async function normalize(input, format) {
  if (format === "json") {
    return {
      alertId: input.id,
      source: input.source,
      timestamp: input.time,
      severity: input.level,
      description: input.msg
    };
  }

  if (format === "xml") {
    const x = await xml2js.parseStringPromise(input);
    const a = x.alert;
    return {
      alertId: a.id[0],
      source: a.source[0],
      timestamp: a.timestamp[0],
      severity: a.severity[0],
      description: a.description[0]
    };
  }

  if (format === "csv") {
    const r = csv.parse(input, { columns: true })[0];
    return {
      alertId: r.id,
      source: r.source,
      timestamp: r.timestamp,
      severity: r.severity,
      description: r.description
    };
  }
}
```

```
(async () => {
  console.log(await normalize(
    { id:"1", source:"FW", time:"T1", level:"High", msg:"Test" },
    "json"
  ));

  console.log(await normalize(
    `<alert><id>2</id><source>IDS</source><timestamp>T2</timestamp><severity>Med</severity><description>X</description></alert>`,
    "xml"
  ));

  console.log(await normalize(
    `id,source,timestamp,severity,description
3,AV,T3,Low,Blocked`,
    "csv"
  ));
})();
```

OUTPUT :

{

ALERTID: '1',

SOURCE: 'FW',

TIMESTAMP: 'T1',

SEVERITY: 'HIGH',

DESCRIPTION: 'TEST'

}

{

ALERTID: '2',

SOURCE: 'IDS',

TIMESTAMP: 'T2',

SEVERITY: 'MED',

DESCRIPTION: 'X'

}

{

ALERTID: '3',

SOURCE: 'AV',

TIMESTAMP: 'T3',

SEVERITY: 'LOW',

DESCRIPTION: 'BLOCKED'

}

OBSERVATION :

1.AI ANALYSES XML, CSV, JSON ALERT SAMPLES TO FIND COMMON FIELDS.

2.DETECTS NAMING CONFLICTS AND RESOLVES THEM TO ONE STANDARD KEY SET.

3.NORMALIZES TIMESTAMP FORMATS INTO ISO 8601.

4.DEFINES UNIFIED SCHEMA WITH CONSISTENT SEVERITY, SOURCE, AND METADATA.

5.GENERATES MAPPING RULES FROM EACH ORIGINAL FORMAT ⟶ UNIFIED SCHEMA.

• **TASK 2**: WRITE ADAPTOR LIBRARY AND TESTS.

PROMPT :

"IMPLEMENT ADAPTORS THAT CONVERT XML/JSON/CSV ALERT FORMATS INTO THE UNIFIED ALERT SCHEMA.

PROVIDE:

* PARSER FUNCTIONS

* VALIDATION LOGIC

* ERROR HANDLING

ALSO GENERATE UNIT TESTS FOR EACH INPUT FORMAT."

CODE :

```
const xml2js = require("xml2js");
const csv = require("csv-parse/sync");

const ok = a => (["alertId","source","timestamp","severity","description"]
  .forEach(f => { if (!a[f]) throw f; }), a);

const fromJSON = j => ok({
  alertId:j.id, source:j.source, timestamp:j.time,
  severity:j.level, description:j.msg
});

const fromXML = async x => {
  const a = (await xml2js.parseStringPromise(x)).alert;
  return ok({
    alertId:a.id[0], source:a.source[0], timestamp:a.timestamp[0],
    severity:a.severity[0], description:a.description[0]
  });
};

const fromCSV = c => {
  const r = csv.parse(c, {columns:true})[0];
  return ok({
    alertId:r.id, source:r.source, timestamp:r.timestamp,
    severity:r.severity, description:r.description
  });
};
(async ()=>{
  console.log(fromJSON({id:"1",source:"FW",time:"T1",level:"High",msg:"X"}));
  console.log(await fromXML(`<alert><id>2</id><source>IDS</source><timestamp>T2</timestamp><severity>M</severity><description>Y</description></alert
  console.log(fromCSV(`id,source,timestamp,severity,description\n3,AV,T3,Low,Z`));
})();
```

OUTPUT :

{

ALERTID: '1',

SOURCE: 'FW',

TIMESTAMP: 'T1',

SEVERITY: 'HIGH',

DESCRIPTION: 'X'

}

{

ALERTID: '2',

SOURCE: 'IDS',

TIMESTAMP: 'T2',

SEVERITY: 'M',

DESCRIPTION: 'Y'

}

{

ALERTID: '3',

SOURCE: 'AV',

TIMESTAMP: 'T3',

SEVERITY: 'LOW',

DESCRIPTION: 'Z'

}

OBSERVATION :

1.AI BUILDS MODULAR ADAPTORS (XML/CSV/JSON) USING PARSING LIBRARIES.

2.APPLIES MAPPING RULES TO NORMALIZE ALL INPUTS.

3.VALIDATES CONVERSIONS AGAINST THE UNIFIED SCHEMA.

4.CREATES JEST TESTS WITH MULTIPLE INPUT SAMPLES.

5.TESTS ERROR HANDLING FOR MALFORMED OR MISSING FIELDS.