What is Utility Layer?

In automation architecture, a **Utility Layer** is:

A centralized reusable component that contains common logic shared across multiple modules.

In your framework:

```
API Layer → calls → Utility Layer → sends actual HTTP request
```

Instead of writing HTTP logic in:

- PostApi.js
- GetApi.js
- PutApi.js
- DeleteApi.js

You centralized everything inside:

```
ApiMethods.js
```

This follows:

- ☑ DRY Principle
- ☑ Separation of Concerns
- ☑ Clean Architecture
- ☑ Reusability Pattern

📦 File: ApiMethods.js

Let’s break it into logical sections.

1️⃣ Imports Section

```
const supertest = require("supertest");
```

What it does:

- Imports Supertest library.
- Supertest is used to send HTTP requests.

Why?

Instead of using axios directly, supertest:

- Works well for testing APIs
- Provides built-in assertions support
- Simplifies request handling

```
const Ajv = require("ajv");
```

What it does:

Imports JSON schema validator.

Why?

To validate response structure.

This ensures:

- API contract correctness
- Prevents silent schema breaking

```
const endpoints = require("../config/endpoints");
```

Imports API URLs from config.

This avoids hardcoding URLs.

```
const payloads = require("../test-data/payloads");
```

Imports test JSON payloads.

Keeps test data separate from logic.

```
const ajv = new Ajv({ allErrors: true, strict: false });
```

Explanation:

- `allErrors: true`
  → Shows all schema errors, not just first one.
- `strict: false`
  → Disables strict mode validation.

Creates a reusable validator instance.

---

2 Class Declaration

```
class ApiMethods {
```

Defines a class.

Why class?

Because:

- We need instance-level configuration.
- We need reusable methods.
- We want OOP abstraction.

---

3 Constructor

```
constructor() {
```

Runs when object is created:

```
this.api = new ApiMethods();
```

```
this.defaultHeaders = {
    "Content-Type": "application/json",
    "Accept": "application/json"
};
```

Purpose:

Sets default headers for every request.

Why?

So API classes don't have to manually set headers each time.

Centralization reduces duplication.

---

4 Token Generator Method

```
async tokengenerator() {
```

Async because it makes API call.

```
const response = await this.request({
```

Calls internal request engine.

```
method: "POST",
url: endpoints.url,
endpoint: endpoints.Tokengenerator,
body: payloads.createtoken,
expectedStatus: 200
```

What happens?

It sends:

```
POST https://restful-booker.herokuapp.com/auth
```

With login payload.

```
const tokenId = response.body.token;
```

Extracts token from response JSON.

```
if (!tokenId) {
    throw new Error("Token not found in response");
}
```

Defensive programming.

Prevents silent failure.

```
return tokenId;
```

Returns authentication token.

---

5 Status Validation

```
validateStatus(response, expectedStatus, method, endpoint) {
```

This is a helper function.

```
if (response.status !== expectedStatus) {
```

Checks if actual status matches expected.

If not, throws detailed error:

- Method
- Endpoint
- Expected
- Actual
- Response body

Why?

So debugging becomes easy.

---

6 Schema Validation

```
validateSchema(schema, responseBody, method, endpoint) {
```

Purpose:
Validate JSON structure.

```
const validate = ajv.compile(schema);
```

Compiles schema into validation function.

```
const valid = validate(responseBody);
```

Runs validation.

If invalid:

```
throw new Error(...)
```

Shows validation errors.

Why important?

Even if status is 200,
API might return wrong structure.

Schema validation protects against that.

---

7 Core Engine – request()

This is the MOST IMPORTANT method.

```
async request({
    method,
    url,
    endpoint,
    token = null,
    body = {},
    queryParams = {},
    headers = {},
    expectedStatus = 200,
    schema = null
})
```

What is this?

Destructured object parameter.

Instead of:

```
request(method, url, endpoint, body, token)
```

You use named parameters.

Why?

- Improves readability
- Allows optional fields
- Prevents parameter order issues

Step 1: Base URL Validation

```
if (!url) throw new Error("Base URL is required");
```

Defensive check.

Step 2: Construct Full URL

```
const fullUrl = `${url}${endpoint}`;
```

Concatenates base URL + endpoint.

Example:

```
https://restful-booker.herokuapp.com/booking
```

Step 3: Initialize Supertest

```
const request = supertest(url);
```

Creates HTTP client instance.

Step 4: Switch Based on HTTP Method

```
switch (method.toUpperCase())
```

Ensures case-insensitive method handling.

GET

```
req = request.get(endpoint).query(queryParams);
```

- Sends GET
- Appends query parameters

POST

```
req = request.post(endpoint).send(body);
```

- Sends JSON body

PUT

```
req = request.put(endpoint).send(body);
```

DELETE

```
req = request.delete(endpoint);
```

Step 5: Merge Headers

```
const finalHeaders = {
    ...this.defaultHeaders,
    ...headers
};
```

Spread operator merges:

- Default headers
- Custom headers

Custom overrides default.

```
req.set(finalHeaders);
```

Attaches headers to request.

Step 6: Token Handling

```
if (token) {
    req.set("Cookie", `token=${token}`);
}
```

Why Cookie?

Restful Booker expects token as cookie.

This abstracts authentication logic.

Step 7: Execute Request

```
const response = await req;
```

Waits until server responds.

Step 8: Validate Status

```
this.validateStatus(...)
```

Ensures response correctness.

Step 9: Validate Schema

```
this.validateSchema(...)
```

Optional schema validation.

Step 10: Return Response

```
return response;
```

Returns full response object.

---

8 Deep Clone

```
async deepClone(obj) {
    return JSON.parse(JSON.stringify(obj));
}
```

Why?

To prevent modifying original payload.

If we modify base payload,
future tests may fail.

---

9 buildPayload()

```
async buildPayload(basePayload, overrides = {})
```

Step 1:
Clone base payload.

Step 2:
Loop through override keys.

```
clone[key] = overrides[key];
```

Step 3:
Return new object.

🧠 Theoretical Design Concepts Used

Your Utility layer demonstrates:

1️⃣ Abstraction

API layer doesn't know about:

- Headers
- Supertest
- Token
- Validation

2️⃣ Encapsulation

All HTTP logic hidden inside one class.

3️⃣ Single Responsibility Principle

ApiMethods handles:

- HTTP
- Validation
- Payload building

API classes handle:

- Business logic

4️⃣ Defensive Programming

- URL check
- Booking ID check
- Token existence check

5️⃣ Reusability

All APIs use same request engine.

🔥 Why Utility Layer is Most Important?

If tomorrow:

- You switch from Supertest → Axios
- You add logging
- You add retry mechanism
- You add timeout
- You add proxy
- You add environment switching

You modify ONLY:

`ApiMethods.js`

Everything else stays same.

That is clean architecture.

🎯 Final Conceptual Flow

```
Step Definition
      ↓
POSTAPI / GETAPI / PUTAPI / DELETEAPI
      ↓
ApiMethods.request()
      ↓
Supertest
      ↓
Server
      ↓
Validation
      ↓
Return response
```