# ✅ 🔥 Complete Summary of Microservice + FeignClient + SCG + Aggregator Flow

## 🎯 ****Your Scenario Setup****

You have **3 functional microservices** and 1 aggregator:

| **Service Name** | **Description** | **Purpose** |
| --- | --- | --- |
| **StudentService** | Handles student info | Takes studentId, returns name, age, aadharId, collegeId |
| **CollegeService** | Handles college info | Takes collegeId, returns college name and address |
| **BGVService** | Govt. background check | Takes aadharId, returns "GOOD"/"BAD" |
| **AggregatorService** | Combines all above | Takes studentId, calls others internally and builds full response |

## ✅ REST ENDPOINT Requirement

You want to call just **one endpoint**:

GET /student-profile/{studentId} --> eila hit chesinappudu below responces ravali

And get a **combined response** like this: - - > e responces anedi all microservices anedi combined avvi vasthundi

{

"studentId": 101,

"name": "Ravi",

"age": 22,

"aadharId": "XXXX-XXXX-XXXX",

"college": {

"collegeId": 10,

"collegeName": "NIT Delhi",

"collegeAddress": "New Delhi"

},

"bgvStatus": "GOOD"

}

## ✅ Technology Involved

| **Tool** | **Used For** |
| --- | --- |
| @FeignClient | Microservice-to-microservice internal calls |
| Spring Cloud Gateway (SCG) | Common external entry point (like API gateway) |
| Aggregator Service | Business logic + Combining multiple service data |
| Eureka (Optional) | Service registry and discovery |
| Spring Boot | Each service is its own Spring Boot project |

## ⚙️ HOW REQUEST FLOWS END-TO-END

### Step-by-step Request Flow

Client calls:

→ http://localhost:9090/student-profile-api/student-profile/101 ← (SCG)

SCG routes to:

→ AggregatorService (student-profile-service)

AggregatorService does:

→ Calls StudentService (via FeignClient): /api/student/101

→ Gets aadharId and collegeId from response

→ Calls CollegeService: /api/college/{collegeId}

→ Calls BGVService: /api/bgv-status/{aadharId}

Combines all → builds final response → returns to client.

## 📦 INDIVIDUAL MICROSERVICE URLS (Internal)

| Service | FeignClient Call (Internal URL) | Returns |
| --- | --- | --- |
| StudentService | /api/student/{id} | studentId, name, age, aadharId, collegeId |
| CollegeService | /api/college/{collegeId} | collegeName, address |
| BGVService | /api/bgv-status/{aadharId} | "GOOD" or "BAD" |

**NOTE:** These are used **internally by FeignClients** — they are NOT exposed to external users.

## 🌐 EXTERNAL URL via SCG

Assume SCG runs on **port 9090**. - - -> Spring Cloud Gate Way anedi 9090 port mida run avuthundi

| Call | Description |
| --- | --- |
| GET http://localhost:9090/student-profile-api/student-profile/101 | Client hits this |
| SCG route maps student-profile-api/\*\* to AggregatorService |  |
| Inside Aggregator, FeignClients call other services (student, college, bgv) |  |

## 💡 Clarifying Your Confusion

### ❓ Q: Can SCG combine responses from multiple services?

**No**, SCG is only for:

Routing external calls to internal microservices

Handling request/response filters, auth, headers, logging, rate limiting

NOT used to **aggregate** data from multiple microservices

🔁 Example:

Client → SCG → /student-profile-api/student-profile/101 → AggregatorService

Conclusion : SCG anedi anni micro-services nuodi responces ni combined or pass cheyadu edi just annitiki common ga oka URL port ni eisthundi : vachhe request ni route chesthundi kavali ante request and responces ni Route chese mundu Authenticate chesi Route chesukovachhu.

### ❓ Q: Can FeignClient be used to call other services from Aggregator?

**✅ Yes. That's the main role of Feign.**

Example:

@FeignClient(name = "student-service")

public interface StudentClient {

@GetMapping("/api/student/{id}")

StudentDTO getStudent(@PathVariable("id") Long id);

}

In AggregatorService, you inject and call it:

StudentDTO student = studentClient.getStudent(studentId);

Conclusion : Oka micro-services nuodi einko micro-services call avvali ante @FeignClient is must

### ❓ Q: Why do we need an AggregatorService?

**Because your use case needs multiple microservices' data merged into one response.**

You need to:

Call student-service → get collegeId & aadharId

Call college-service with collegeId

Call bgv-service with aadharId

Combine all → respond

Only AggregatorService can do this — not SCG, not FeignClient alone.

Conclusion : AggregatorService anedi anni microservices nuodi responces collect cheyataniki vatini combind cheyataniki AggregatorService lo code rayali

## ✅ When to Use What — Summary Table

| **Scenario** | **Use SCG** | **Use FeignClient** | **Use Aggregator** |
| --- | --- | --- | --- |
| Route to internal service using single port | ✅ | ❌ | ❌ |
| One service calling another | ❌ | ✅ | ❌ |
| Combine response from multiple services | ❌ | ✅ (inside) | ✅ |
| Apply auth, logging, security filters globally | ✅ | ❌ | ❌ |
| Expose one endpoint to client, handle business logic | ✅ | ✅ | ✅ |

## 📘 Bonus: SCG Routing Example (application.yml)

spring:

cloud:

gateway:

routes:

- id: student-profile

uri: lb://student-profile-service

predicates:

- Path=/student-profile-api/\*\*

filters:

- StripPrefix=1

- id: student-service

uri: lb://student-service

predicates:

- Path=/student-service-api/\*\*

filters:

- StripPrefix=1

- id: college-service

uri: lb://college-service

predicates:

- Path=/college-service-api/\*\*

filters:

- StripPrefix=1

- id: bgv-service

uri: lb://bgv-service

predicates:

- Path=/bgv-service-api/\*\*

filters:

- StripPrefix=1

With above setup, all services can be accessed via:

localhost:9090/student-service-api/\*\*

localhost:9090/college-service-api/\*\*

localhost:9090/bgv-service-api/\*\*

localhost:9090/student-profile-api/\*\*

But **clients** will only use:

GET /student-profile-api/student-profile/{studentId}

because it returns everything they need.

## 🏁 Final Output

🎯 You created an **Aggregator endpoint** that:

Accepts studentId

Internally calls multiple services using @FeignClient

Combines all responses

Is exposed externally via **SCG** under localhost:9090

You now understand:

✅ Why SCG doesn't combine responses

✅ Why Feign is only internal

✅ Why AggregatorService is necessary

✅ When and how to use each in real-time architecture

| Component | Coding Required? | Role in Microservices | Real-Time Usage |
| --- | --- | --- | --- |
| Spring Cloud Gateway (SCG) | ❌ No coding, only configuration (in application.yml) | Acts as a single entry point (router/load balancer) to access all microservices through one port | Used to simplify external calls, apply global filters (auth, rate limiting, etc.) |
| @FeignClient | ✅ Yes, used in code | Makes internal microservice-to-microservice REST calls using interfaces | Used inside Aggregator or any service to call another microservice (cleaner than RestTemplate) |
| Aggregator Service | ✅ Yes, full code needed | Central business logic that combines data from multiple microservices using FeignClient calls | Used when a client wants one combined response from multiple services |

### ✅ FINAL CONCLUSION

Final Points :

@FeignClient : one micro-services to another micro-services ni call cheyataniki vadatahm ani thelusu

Spring Cloud Gate Way : Starting lo SCG anedi multiple microservices ni call chesi responces eisthundi anukunnam

But -> edi just request ni and response ni route chesthundi adi kuda single port mida

AggregatorService : Starting lo AggregatorService anedi separate coding ani migilina topics tho edi work avuthundi anukunnam

But -> edi a two topics ni use chesukoni anni microservices ni responces cobine chesi thisukovataniki logic eindulo rasukovali

Means edi just topic name anthe

========== Main Concept calling External API in existing Code ============

How to Implement and how to interact with External API’s into existing URL ?

To Interact with External API’s we have 2 ways

1. RestTemplate 2) WebClient

❓ **What is the difference between** RestTemplate **and** WebClient**, and when should we use which one?**

Let’s break it down **clearly**, with **real-time examples and use cases** to help you **understand and remember easily**.

## 1. BASIC DIFFERENCE

| **Feature** | **RestTemplate (Old)** | **WebClient (New)** |
| --- | --- | --- |
| **Type** | Synchronous (blocking) | Asynchronous (non-blocking) |
| **Spring Version** | Used in Spring 4 & 5 | Introduced in Spring 5+ |
| **Replaced?** | Deprecated (slowly) | ✅ Recommended going forward |
| **Performance** | Slower (waits for response) | Faster (doesn’t block thread) |
| **Thread Model** | Each call blocks a thread | Uses reactive, event-driven |

## 🧪 2. SIMPLE CODE EXAMPLES

### ✅ RestTemplate Example (Blocking):

RestTemplate restTemplate = new RestTemplate();

String url = "http://localhost:8081/student/1";

Student student = restTemplate.getForObject(url, Student.class);

System.out.println(student.getName());

🔁 Waits until the response comes

❌ Not suitable for high-traffic systems

### ✅ WebClient Example (Non-blocking):

WebClient webClient = WebClient.create();

String url = "http://localhost:8081/student/1";

webClient.get()

.uri(url)

.retrieve()

.bodyToMono(Student.class)

.subscribe(student -> System.out.println(student.getName()));

✅ Doesn’t block the thread → continues other work while waiting

✅ Good for modern, reactive applications

## 🔍 3. REAL-TIME SCENARIO EXAMPLES

### ****Scenario 1: Internal Microservice-to-Microservice Call****

**Use** FeignClient (not RestTemplate or WebClient)

Because it’s easier, declarative, and works with Eureka, Load Balancer

### ****Scenario 2: Calling 3rd Party API (Small traffic, blocking is OK)****

**Use** RestTemplate

✅ Simpler code

🧠 If you’re calling an external API occasionally (ex: fetch BGV status once), RestTemplate is **fine**.

### ****Scenario 3: Calling 3rd Party API (Heavy traffic, parallel calls)****

**Use** WebClient

✅ Scalable, modern apps

Example: When calling **multiple APIs** (weather, bgv, SMS service) together in parallel and combining results quickly

Mono<Student> studentMono = webClient.get().uri("/student/1").retrieve().bodyToMono(Student.class);

Mono<College> collegeMono = webClient.get().uri("/college/1").retrieve().bodyToMono(College.class);

Mono.zip(studentMono, collegeMono).subscribe(tuple -> {

Student s = tuple.getT1();

College c = tuple.getT2();

// merge results

});

## 4. Summary: When to Use What

| **Scenario** | **Use** | **Why?** |
| --- | --- | --- |
| Microservice-to-microservice call | @FeignClient | Easy, uses service discovery |
| Call external API, simple requirement | RestTemplate | Simple & easy to use |
| Call external API, high performance | WebClient | Non-blocking & efficient |
| Reactive project (Spring WebFlux) | WebClient only | RestTemplate won’t work here |
| Complex parallel or async calls | WebClient | Designed for async scenarios |

## 🎯 Final Recommendation (Real-Time Practice)

| **Layer** | **Tool to Use** |
| --- | --- |
| Internal Service Call | ✅ @FeignClient |
| External API (BGV, SMS, etc) |  |

✅ RestTemplate if simple

✅ WebClient if modern/high traffic  
| Aggregation Layer | ✅ Combine all responses, handle exceptions

## 📝 Notes for Beginners

✔️ RestTemplate is **blocking** and **easy for beginners**

🚀 WebClient is **non-blocking**, future-proof, but needs some practice

Spring says: “Prefer WebClient for new code.”

========================= Example Code for Both ======================

### / ------------ Common DTO Class (Student.java) ------------

### public class Student {

### private int studentId;

### private String name;

### private int age;

### private String aadharId;

### 

### // Getters and Setters

### }

### // ------------ Using RestTemplate ------------

### @RestController

### @RequestMapping("/resttemplate")

### public class RestTemplateController {

### private final RestTemplate restTemplate;

### public RestTemplateController() {

### this.restTemplate = new RestTemplate();

### }

### @GetMapping("/student/{id}")

### public Student getStudentUsingRestTemplate(@PathVariable int id) {

### String url = "http://localhost:8081/api/student/" + id;

### return restTemplate.getForObject(url, Student.class);

### }

### }

### // ------------ Using WebClient ------------

### @RestController

### @RequestMapping("/webclient")

### public class WebClientController {

### private final WebClient webClient = WebClient.builder().baseUrl("http://localhost:8081").build();

### @GetMapping("/student/{id}")

### public Mono<Student> getStudentUsingWebClient(@PathVariable int id) {

### return webClient.get()

### .uri("/api/student/" + id)

### .retrieve()

### .bodyToMono(Student.class);

### }

### }

### // ------------ Mock Student Service (for testing) ------------

### @RestController

### @RequestMapping("/api/student")

### public class MockStudentService {

### @GetMapping("/{id}")

### public Student getMockStudent(@PathVariable int id) {

### Student student = new Student();

### student.setStudentId(id);

### student.setName("Ravi");

### student.setAge(22);

### student.setAadharId("1234-5678-9012");

### return student;

### }

### }

### ✅ Summary of Use Cases:

| **Feature** | **RestTemplate** | **WebClient** |
| --- | --- | --- |
| Style | Synchronous (blocking) | Asynchronous + Non-blocking (reactive) |
| Real-time usage | Simple internal/external API calls | Modern reactive systems (large scale, async) |
| Best for beginners | ✅ Easier to understand and use | ❌ Needs knowledge of reactive programming |
| Preferred for reactive apps | ❌ Not suitable | ✅ Highly suitable |
| Deprecated? | ✅ Soft-deprecated (Spring recommends WebClient) | ❌ Preferred option in Spring Boot 2+ |

Yes, you are absolutely right — in **Spring Boot (or Java in general)**, you can use URI **or** URL **classes from** java.net **package** to call external APIs. But this approach is **very low-level**, and **not the recommended or modern way** for Spring Boot applications. ?

### Comparison with RestTemplate and WebClient

| **Feature** | **java.net.URI/HttpURLConnection** | **RestTemplate** | **WebClient** |
| --- | --- | --- | --- |
| Abstraction Level | Low (manual coding, error-prone) | Medium (easy to use, blocking) | High (async, non-blocking) |
| Suitable for Spring? | ❌ Not recommended | ✅ Recommended for simple cases | ✅✅ Preferred for modern Spring apps |
| Handles JSON easily? | ❌ Requires manual parsing | ✅ Automatic conversion (Jackson) | ✅ Auto-conversion, streaming |
| Error handling | ❌ Manual | ✅ Built-in | ✅ Advanced |

### ✅ Example using java.net.HttpURLConnection (not recommended)

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.net.HttpURLConnection;

import java.net.URL;

public class BGVCaller {

public String getBGVStatus(int studentId) {

try {

URL url = new URL("http://localhost:3345/BGV-Report-URL/api/student-BGV/getById/" + studentId);

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

conn.setRequestMethod("GET");

BufferedReader in = new BufferedReader(

new InputStreamReader(conn.getInputStream())

);

String inputLine;

StringBuilder content = new StringBuilder();

while ((inputLine = in.readLine()) != null) {

content.append(inputLine);

}

in.close();

conn.disconnect();

return content.toString(); // should parse to get "GOOD"/"BAD"

} catch (Exception e) {

return "UNKNOWN";

}

}

}

### Why this is ****not preferred**** in Spring Boot?

❌ No automatic JSON to Java object conversion.

❌ No built-in support for timeouts, retries, headers.

❌ Very verbose.

❌ Not test-friendly or maintainable.

### ✅ What to use instead?

**Use** RestTemplate if you are writing a simple sync call.

**Use** WebClient if you want non-blocking and modern reactive support.

Avoid java.net.URL or URI based manual calls in Spring Boot unless you're doing something extremely custom.