How do you utilize overloading concepts while implementing apis ?

This is a trick question. While Java supports method overloading, **REST APIs (HTTP) do not support overloading** in the same way. You cannot have two endpoints with the same URL `POST /users` that differ only by parameters.

Here is the correct way to frame your answer:

1. The Challenge (Java Overloading vs. REST)

"In Java, I can have `createUser(String name)` and `createUser(String name, int age)`.
But in REST, I cannot expose two POST endpoints on `/users` just because the JSON body is different. The URL must be unique."

2. How I "Simulate" Overloading in REST APIs

"Instead of overloading the *endpoint*, I overload the **Service Method** and use a flexible **DTO** or **Query Parameters** in the Controller."

Scenario 1: Search API (Query Params)

"I want to search users by `name` OR by `email` OR by `role`.
Instead of 3 different endpoints, I use one endpoint with optional parameters."

Controller:

```
@GetMapping("/users")
public ResponseEntity<List<User>> searchUsers(
    @RequestParam(required = false) String name,
    @RequestParam(required = false) String email) {

    // Service handles the logic (Simulated Overloading)
    if (email != null) return service.searchByEmail(email);
    if (name != null) return service.searchByName(name);
    return service.getAllUsers();
}
```

Scenario 2: Create API (Flexible DTO)

"I want to create a user with just `email` (minimal) OR with `email + address` (full).
I use a single `CreateUserRequest` DTO where optional fields can be null."

Controller:

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody CreateUserRequest request) {
    // The Service has overloaded methods!
    if (request.getAddress() == null) {
        return service.createUser(request.getEmail()); // Calls Method A
    } else {
        return service.createUser(request.getEmail(), request.getAddress()); // Calls Met
    }
}
```

3. True Overloading in Service Layer

"While the Controller remains simple (one public entry point), my **Service Layer** utilizes Java overloading extensively to keep business logic clean."

```
@Service
public class UserService {
    // Method 1
    public User createUser(String email) {
        return createUser(email, "DEFAULT_ADDRESS"); // Chains to Method 2
    }

    // Method 2 (The one doing the work)
    public User createUser(String email, String address) {
        User user = new User(email, address);
        return repo.save(user);
    }
}
```

Summary Answer

"I don't overload the API endpoints (URLs) directly because REST doesn't support it. Instead, I use **optional query parameters** or **flexible DTOs** in the Controller to capture different input variations. Then, inside the **Service layer**, I call overloaded Java methods to handle the specific business logic cleanly."

⁂

1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/429036f4-3bc3-4a7a-8c6f-90246e73b9b9/image.jpg
2. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/24f18c0a-b56e-4862-a619-e2959036a5c2/image.jpg

3. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/97e9ed86-ed11-4035-a18b-1eb05c1bab4e/image.jpg
4. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/87f62423-96f2-4071-9802-8f6699e0ecd8/image.jpg