I understand your confusion, and I appreciate you pointing this out. Let's clarify why each feature catches `HttpServerErrorException` instead of `HttpClientErrorException`, and address the concern about the `handleOnboardingFailureException` logic in the `GlobalExceptionHandler`. Your observation suggests we need to ensure the exception handling aligns with the HTTP error types (server vs. client errors) and that the logic in `@ExceptionHandler` makes sense in this context.

Why `HttpServerErrorException` and Not `HttpClientErrorException`?

In the feature executors (`AccountCreationExecutor`, `OnlineBankingExecutor`), we're catching `HttpServerErrorException` because:

- **`HttpServerErrorException`**: Represents server-side errors (HTTP 5xx status codes, e.g., 500 Internal Server Error, 503 Service Unavailable). These indicate issues with the downstream API (e.g., the account creation or online banking service is down or malfunctioning).

- **`HttpClientErrorException`**: Represents client-side errors (HTTP 4xx status codes, e.g., 400 Bad Request, 404 Not Found). These typically indicate a problem with the request itself (e.g., invalid input), not the server.

Current Context

- The downstream APIs (`http://downstream-api/accounts/create`, `http://downstream-api/online-banking/enable`) are external services. Failures like HTTP 500 or 503 are server-side issues beyond the control of your application, hence `HttpServerErrorException` is appropriate.

- We're assuming that client errors (e.g., 400 Bad Request) are less likely from these APIs in this flow, or they'd be caught earlier (e.g., in input validation). If a client error occurs, it's not currently handled by the feature executors, which could be a gap.

Your Concern

- If every feature only catches `HttpServerErrorException`, what happens with `HttpClientErrorException` (e.g., HTTP 400)? It would bubble up unhandled, potentially caught by the generic `Exception` handler, leading to an inconsistent response.

- The `handleOnboardingFailureException` checks for "HTTP 500" in the message to return `500 Internal Server Error`, but this string-based check feels brittle and doesn't account for other server errors (e.g., 503) or client errors.

Clarifying and Fixing the Approach

To resolve this:

1. **Handle Both Exception Types**: Feature executors should catch both `HttpServerErrorException` (5xx) and `HttpClientErrorException` (4xx) to differentiate server vs. client errors.

2. **Improve Exception Handling**: Pass the exception type or status code through `FeatureExecutionException` to `OnboardingFailureException`, avoiding string parsing in `@ExceptionHandler`.

3. **Dynamic Partial Success**: Ensure partial success detection remains robust with dynamic features, while aligning HTTP status codes correctly.

Here's the revised solution:

Updated Example

1. Update Exceptions

Enhance `FeatureExecutionException` to carry the HTTP status code.

```java
public class FeatureExecutionException extends RuntimeException {
    private final int httpStatusCode;

    public FeatureExecutionException(String message, int httpStatusCode, Throwable cause) {
        super(message, cause);
        this.httpStatusCode = httpStatusCode;
    }

    public int getHttpStatusCode() {
        return httpStatusCode;
    }
}
```

```
public class OnboardingFailureException extends RuntimeException {
    private final String errorCode;
    private final int httpStatusCode;

    public OnboardingFailureException(String message, String errorCode, int httpStatusCode) {
        super(message);
        this.errorCode = errorCode;
        this.httpStatusCode = httpStatusCode;
    }

    public String getErrorCode() {
        return errorCode;
    }

    public int getHttpStatusCode() {
        return httpStatusCode;
    }
}
```

- **Change**: Added `httpStatusCode` to exceptions for precise status handling.

2. Update Feature Executors
Catch both `HttpServerErrorException` and `HttpClientErrorException`.

```java
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.HttpServerErrorException;
import org.springframework.web.client.RestTemplate;
import org.springframework.stereotype.Service;
```

```
public interface FeatureExecutor {
  FeatureResponse execute(CustomerRequest request, Map<String, FeatureResponse>
previousResults) throws FeatureExecutionException;
  String getFeatureName();
}


@Service
public class AccountCreationExecutor implements FeatureExecutor {
  private final RestTemplate restTemplate;


  public AccountCreationExecutor(RestTemplate restTemplate) {
    this.restTemplate = restTemplate;
  }


  @Override
  public FeatureResponse execute(CustomerRequest request, Map<String, FeatureResponse>
previousResults) throws FeatureExecutionException {
    try {
      String url = "http://downstream-api/accounts/create?name=" + request.getName() + "&id=" +
request.getIdNumber();
      ResponseEntity<String> apiResponse = restTemplate.getForEntity(url, String.class);
      return new AccountCreationResponse(apiResponse.getBody(), "CREATED", "2025-03-
24T10:00:00Z");
    } catch (HttpServerErrorException e) {
      throw new FeatureExecutionException("Account creation failed: " + e.getMessage(),
e.getStatusCode().value(), e);
    } catch (HttpClientErrorException e) {
      throw new FeatureExecutionException("Account creation invalid request: " + e.getMessage(),
e.getStatusCode().value(), e);
    }
  }


  @Override
```

```
    public String getFeatureName() {
        return "accountCreation";
    }
}

@Service
public class OnlineBankingExecutor implements FeatureExecutor {
    private final RestTemplate restTemplate;

    public OnlineBankingExecutor(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @Override
    public FeatureResponse execute(CustomerRequest request, Map<String, FeatureResponse>
previousResults) throws FeatureExecutionException {
        AccountCreationResponse accountResponse = (AccountCreationResponse)
previousResults.get("accountCreation");
        if (accountResponse == null) {
            throw new FeatureExecutionException("Online banking requires account creation to succeed
first", 400, null);
        }
        String accountNumber = accountResponse.getAccountNumber();
        String url = "http://downstream-api/online-banking/enable?account=" + accountNumber +
"&id=" + request.getIdNumber();
        try {
            restTemplate.postForEntity(url, null, String.class);
            return new OnlineBankingResponse(accountNumber, "ENABLED", "2025-03-24T10:01:00Z");
        } catch (HttpServerErrorException e) {
            throw new FeatureExecutionException("Online banking failed: " + e.getMessage(),
e.getStatusCode().value(), e);
        } catch (HttpClientErrorException e) {
            throw new FeatureExecutionException("Online banking invalid request: " + e.getMessage(), e.getStatusCode().value(), e);
        }
    }

    @Override
    public String getFeatureName() {
        return "onlineBanking";
    }
}
```

- **Change**: Catch both exception types, propagate the HTTP status code.

3. Update `NaturalPersonServiceImpl`
Pass HTTP status through exceptions.

```java
import org.springframework.stereotype.Service;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service
public class NaturalPersonServiceImpl {

    private final Map<String, List<FeatureExecutor>> productFeatureMap;
    private final RestTemplate restTemplate;

    public NaturalPersonServiceImpl(Map<String, List<FeatureExecutor>> productFeatureMap, RestTemplate restTemplate) {
        this.productFeatureMap = productFeatureMap;
        this.restTemplate = restTemplate;
    }

    public OnboardingResponse onboardNaturalPerson(CustomerRequest request, String
productType) {
        checkMinimumAge(request.getIdNumber());
        verifyIdentity(request.getIdNumber());

        List<FeatureExecutor> features = productFeatureMap.getOrDefault(productType, List.of());
        if (features.isEmpty()) {
            throw new IllegalArgumentException("No features configured for product type: " +
productType);
        }

        Map<String, FeatureResponse> featureResponses = new HashMap<>();
        String errorMessage = null;
        int successfulFeatures = 0;

        for (FeatureExecutor feature : features) {
            try {
                FeatureResponse response = feature.execute(request, featureResponses);
                featureResponses.put(feature.getFeatureName(), response);
                successfulFeatures++;
            } catch (FeatureExecutionException e) {
                errorMessage = e.getMessage();
                int statusCode = e.getHttpStatusCode();
                if (successfulFeatures > 0) {
                    // Partial success
                    return new OnboardingResponse(featureResponses, errorMessage);
                } else {
                    // Total failure
                    String errorCode = (statusCode >= 500 && statusCode < 600) ? "SERVER_ERROR" :
"ONBOARDING_FAILED";
                    throw new OnboardingFailureException(errorMessage, errorCode, statusCode);
                }
            }
        }

        return new OnboardingResponse(featureResponses, null);
    }

    private void checkMinimumAge(String idNumber) {
        int birthYear = Integer.parseInt(idNumber.substring(0, 2)) + 2000;
        int currentYear = 2025;
        int age = currentYear - birthYear;
        if (age < 18) {
            throw new MinimumAgeRequirementException("Customer with ID " + idNumber + " is under
18 (age: " + age + ")");
        }
    }

    private void verifyIdentity(String idNumber) {
        String url = "http://identity-api/verify?id=" + idNumber;
        try {
            ResponseEntity<String> response = restTemplate.getForEntity(url, String.class);
            if (!"VALID".equals(response.getBody())) {
                throw new IdentityVerificationException("Identity verification failed for ID: " + idNumber);
            }
        } catch (HttpClientErrorException e) {
            throw new IdentityVerificationException("Identity verification failed due to API error: " +
e.getMessage());
        }
    }
```

}
~~~

- **Change**: Use `httpStatusCode` to determine `errorCode` (`SERVER_ERROR` for 5xx, `ONBOARDING_FAILED` for others).

4. Update `GlobalExceptionHandler`

Use `httpStatusCode` instead of string parsing.

```java
import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.*;

@ControllerAdvice

public class GlobalExceptionHandler {

    @ExceptionHandler(OnboardingFailureException.class)

    public ResponseEntity<ErrorResponse> handleOnboardingFailureException(OnboardingFailureException e) {

        int statusCode = e.getHttpStatusCode();

        HttpStatus httpStatus = (statusCode >= 500 && statusCode < 600) ? HttpStatus.INTERNAL_SERVER_ERROR : HttpStatus.BAD_REQUEST;

        return new ResponseEntity<>(new ErrorResponse(e.getErrorCode(), e.getMessage()), httpStatus);

    }

    @ExceptionHandler(IdentityVerificationException.class)

    public ResponseEntity<ErrorResponse> handleIdentityVerificationException(IdentityVerificationException e) {

        return new ResponseEntity<>(new ErrorResponse("IDENTITY_VERIFICATION_FAILED", e.getMessage()), HttpStatus.FORBIDDEN);

    }
~~~

```
    @ExceptionHandler(MinimumAgeRequirementException.class)
    public ResponseEntity<ErrorResponse> handleMinimumAgeRequirementException(MinimumAgeRequirementException e) {
        return new ResponseEntity<>(new ErrorResponse("MINIMUM_AGE_NOT_MET", e.getMessage()), HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(Exception e) {
        return new ResponseEntity<>(new ErrorResponse("INTERNAL_ERROR", "Unexpected error: " + e.getMessage()), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

- **Change**: Removed string parsing (`message.contains("HTTP 500")`); uses `httpStatusCode` for precise status determination.

5. Test the Flows
- **Success (200 OK)**:
  - **Response**:
    ```json
    {
      "featureResponses": {
        "accountCreation": { "accountNumber": "1234567890", "status": "CREATED", "creationTimestamp": "2025-03-24T10:00:00Z" },
        "onlineBanking": { "accountNumber": "1234567890", "status": "ENABLED", "activationTimestamp": "2025-03-24T10:01:00Z" }
      },
      "errorMessage": null
    }
    ```

- **Partial Success (206 Partial Content)**:
  - **Scenario**: Online banking fails with HTTP 503.
  - **Response**:
    ```json
    {
      "featureResponses": {
        "accountCreation": { "accountNumber": "1234567890", "status": "CREATED", "creationTimestamp": "2025-03-24T10:00:00Z" }
      },
      "errorMessage": "Online banking failed: 503 Service Unavailable"
    }
    ```

- **Total Failure (HTTP 500)**:
  - **Response**:
    ```json
    {
      "errorCode": "SERVER_ERROR",
      "message": "Account creation failed: 500 Internal Server Error"
    }
    ```
    - HTTP Status: `500 Internal Server Error`

- **Total Failure (HTTP 400)**:
  - **Response**:
    ```json
    {
      "errorCode": "ONBOARDING_FAILED",
      "message": "Account creation invalid request: 400 Bad Request"
    }
    ```

```

   - HTTP Status: `400 Bad Request`

---

### Clarifications

1. **Why Both Exceptions?**:

   - `HttpServerErrorException` (5xx): Handles server issues (e.g., 500, 503) from downstream APIs.

   - `HttpClientErrorException` (4xx): Handles client errors (e.g., 400) if the downstream API rejects the request due to invalid data. This ensures all REST exceptions are caught.

2. **Improved Handler Logic**:

   - Instead of parsing `message.contains("HTTP 500")`, we use `httpStatusCode` to check if the status is in the 5xx range (500–599).

   - This is more robust, covering all server errors (e.g., 503, 502), not just 500, and avoids brittle string matching.

3. **Dynamic Features**:

   - The solution remains flexible: partial success is detected by counting successful features, regardless of order or type.

---

### Conclusion

- **Exception Handling**: Each feature now catches both `HttpServerErrorException` and `HttpClientErrorException`, ensuring all REST errors are handled.

- **Handler Logic**: `handleOnboardingFailureException` uses `httpStatusCode` for accurate status mapping (5xx → 500, 4xx → 400), meeting your requirement of `ErrorResponse` for failures.

- **Dynamic Partial Success**: Works seamlessly with configurable feature lists, returning `OnboardingResponse` only for 200 and 206.

This should now be clearer and more robust. Let me know if you need further explanation or adjustments!
```