Take-Home Assignment: Build a Non-Blocking HTTP Invocation API

Problem Context

Imagine you're building an **internal orchestration API** for a platform that handles **high volumes of incoming HTTP traffic** (e.g., 1000+ concurrent requests). Each incoming request triggers a downstream HTTP call to external services.

A sample implementation uses **synchronous Apache `HttpClient`**, which blocks the thread until the downstream call completes. Under heavy load, this causes **thread exhaustion**, **poor scalability**, and **slow response times**.

Assignment Objective

Design and implement an **asynchronous**, **non-blocking** HTTP invocation API that accepts the following parameters and performs an outbound HTTP call:

```
ApiMethod apiMethod,

RequestDTO requestDTO,

SSLConnectionSocketFactory sslConnectionSocketFactor,

int timeout
```

The goal is to **free the thread** immediately after dispatching the request, and **not wait** for the downstream response to complete synchronously.

Input API

You are expected to expose an API endpoint that accepts a JSON request with the following fields:

```
{

  "apiMethod": "POST",

  "requestDTO": {
    "url": "https://example.com",
    "headerVariables": {
      "Authorization": "Bearer token"
    },
    "params": [
      { "name": "key1", "value": "value1" }
    ],
    "bodyType": "application/json",
    "requestBody": "{\"key\": \"value\"}"
  },
  "timeout": 5000
}
```

(SSL configuration can be mocked if required.)

Requirements

Concurrency

- The API should handle **multiple parallel incoming requests** without blocking.
- Demonstrate that **thread 2 does not wait** for thread 1 to finish when invoking downstream services.

Non-Blocking I/O

- Refactor the `executeTarget()` logic to use **non-blocking** HTTP clients (e.g., `WebClient`, Java 11+ `HttpClient`with `CompletableFuture`, or `Apache HttpAsyncClient`).
- Ensure true asynchronous I/O.

HTTP Method Compatibility

- Your implementation must support: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `OPTIONS`.

Timeout and SSL Support

- Support **per-request timeout**.
- Handle optional **SSL configuration** (if not feasible, stub/mock).

Error Handling

- Capture and propagate downstream HTTP call errors meaningfully.

Extensibility

- Structure the code to allow future enhancements:
  - Retry logic
  - Request/response logging
  - Tracing (e.g., with correlation IDs)
  - Metrics instrumentation

Deliverables

1. **Refactored API source code**, preferably structured as:
   - `RestFactoryAsync` (or similar)
   - Controller that receives incoming HTTP requests with input JSON
2. **Load test scripts** (e.g., JMeter, k6, or Gatling) simulating at least **1000 parallel requests**
3. **Before & after performance benchmarks** with:
   - Avg. response time
   - Max response time
   - Throughput (RPS)
4. **README or comments** explaining:
   - Architecture
   - Tech choices
   - Trade-offs and limitations

Sample Scenario

You receive **1000 concurrent POST requests** to your API, each needing to call an external endpoint. In the existing synchronous model, threads are tied up waiting for responses. In your new implementation, requests should be dispatched instantly, and the system must **continue accepting traffic without blocking**.

Sample Code.

Interface

```
public interface ApiFactory {
   public abstract HttpResponse executeTarget(
           ApiMethod apiMethod,
           RequestDTO requestDTO,
           SSLConnectionSocketFactory sslConnectionSocketFactory,
           int timeout
   ) throws IOException;
}
```

Request DTO class

```
public class RequestDTO {
  private ApiMethod apiMethod;
  private String url;
  private Map<String, String> queryParams;
  private Map<String, String> headerVariables;
  private String bodyType;
  private String requestBody;
  private Map<String, String> pathMap;
  private Map<String, String> formParam;
  private Map<String, String> urlEncodedParam;
  private List<NameValuePair> params;
}
}
```

ApiMethod

```
public enum ApiMethod {
  GET,
  POST,
  PUT,
  DELETE,
  PATCH,
  OPTIONS;

  private ApiMethod() {}
}
  }
}
```

Factory class

```
public class RestFactory implements ApiFactory{

   private static final Logger logger =
LoggerFactory.getLogger(RestFactory.class);

   @Override
   public HttpResponse executeTarget(ApiMethod apiMethod, RequestDTO
requestDTO,
                                     SSLConnectionSocketFactory
sslConnectionSocketFactory, int timeout) throws IOException {
       return switch (apiMethod) {
           case GET -> invokeGet(requestDTO.getUrl(),
requestDTO.getHeaderVariables(),
               sslConnectionSocketFactory, timeout);
           case POST -> invokePost(requestDTO.getUrl(),
requestDTO.getHeaderVariables(),
               requestDTO.getParams(), requestDTO.getBodyType(),
requestDTO.getRequestBody(),
               sslConnectionSocketFactory, timeout);
           case PUT -> invokePut(requestDTO.getUrl(),
requestDTO.getHeaderVariables(),
               requestDTO.getParams(), requestDTO.getBodyType(),
requestDTO.getRequestBody(),
               sslConnectionSocketFactory, timeout);
           case DELETE -> invokeDelete(requestDTO.getUrl(),
requestDTO.getHeaderVariables(),
               requestDTO.getRequestBody(), sslConnectionSocketFactory,
timeout);
           case PATCH -> invokePatch(requestDTO.getUrl(),
requestDTO.getHeaderVariables(),
               requestDTO.getRequestBody(), sslConnectionSocketFactory,
timeout);
           case OPTIONS -> invokeOptions(requestDTO.getUrl(),
requestDTO.getHeaderVariables(),
               sslConnectionSocketFactory, timeout);
       };
   }

   public HttpResponse invokeGet(String url, Map<String, String> headers,
           SSLConnectionSocketFactory sslConnectionFactory, int timeout) throws
IOException {
       HttpClient client = getClient(sslConnectionFactory, timeout);
       HttpGet request = new HttpGet(url);
       addHeaders(request, headers);
//         long startTime = System.currentTimeMillis();
       return client.execute(request);
   }
   public HttpResponse invokePost(String url, Map<String, String> headers,
List<NameValuePair> params,
           String type, String body, SSLConnectionSocketFactory
sslConnectionFactory, int timeout)
           throws IOException {
       HttpClient client = getClient(sslConnectionFactory, timeout);
       HttpPost request = new HttpPost(url);
       updatePostPutRequest(headers, params, type, body, request);
       return client.execute(request);
   }

   public HttpResponse invokePut(String url, Map<String, String> headers,
List<NameValuePair> params,
                                        String type, String body,
SSLConnectionSocketFactory sslConnectionFactory, int timeout)
           throws IOException {

       HttpClient client = getClient(sslConnectionFactory, timeout);
       HttpPut request = new HttpPut(url);
       updatePostPutRequest(headers, params, type, body, request);
       return client.execute(request);
   }
   public HttpResponse invokeDelete(String url, Map<String, String> headers,
String body,
                                           SSLConnectionSocketFactory
sslConnectionFactory, int timeout) throws IOException {
       HttpClient client = getClient(sslConnectionFactory, timeout);
       HttpRequestBase request;
       if(null != body && !body.isEmpty()){
           request = new HttpDeleteWithBody(url);
           addBody(request, body);
       } else {
           request=new HttpDelete(url);
       }
       addHeaders(request, headers);
       return client.execute(request);
   }
   public HttpResponse invokePatch(String url, Map<String, String> headers,
String body,
                                   SSLConnectionSocketFactory
sslConnectionFactory, int timeout) throws IOException {
       HttpClient client = getClient(sslConnectionFactory, timeout);
       HttpPatch request = new HttpPatch(url);
       request.setEntity(new StringEntity(body));
       addHeaders(request, headers);
       return client.execute(request);
   }
   public HttpResponse invokeOptions(String url, Map<String, String> headers,
                                            SSLConnectionSocketFactory
sslConnectionFactory, int timeout) throws IOException {
       HttpClient client = getClient(sslConnectionFactory, timeout);
       HttpOptions request = new HttpOptions(url);
       addHeaders(request, headers);
       return client.execute(request);
   }

   private void updatePostPutRequest(Map<String, String> headers,
List<NameValuePair> params, String type,
           String body, HttpEntityEnclosingRequestBase request) throws
UnsupportedEncodingException {
       if (body == null && type != null) {
           if (type.equals("multi-part")) {
               headers.remove("Content-Type");
               MultipartEntityBuilder entityBuilder =
MultipartEntityBuilder.create();
               entityBuilder.setMode(HttpMultipartMode.BROWSER_COMPATIBLE);
               if(params != null) {
                   for (NameValuePair param : params) {
                       entityBuilder.addTextBody(param.getName(),
param.getValue(), ContentType.DEFAULT_BINARY);
                   }
               }
               request.setEntity(entityBuilder.build());
           } else {
               List<NameValuePair> formParams = new ArrayList<>();
               if(params != null) {
                   for (NameValuePair param : params) {
                       formParams.add(new BasicNameValuePair(param.getName(),
param.getValue()));
                   }
               }
               UrlEncodedFormEntity entity = new
UrlEncodedFormEntity(formParams, Consts.UTF_8);
               request.setEntity(entity);
           }

       } else if(body != null){
           request.setEntity(new StringEntity(body));
       }
       addHeaders(request, headers);
   }

   private HttpClient getClient(SSLConnectionSocketFactory
sslConnectionFactory, int timeout) {
       HttpClient client;
       if (sslConnectionFactory != null) {
           logger.debug("Getting client details");
```

```
            client =
HttpClientBuilder.create().setSSLSocketFactory(sslConnectionFactory)

.setDefaultRequestConfig(requestConfigWithTimeout(timeout)).build();
        } else {
            client =
HttpClientBuilder.create().setDefaultRequestConfig(requestConfigWithTimeout(tim
eout)).build();
        }
        return client;
    }

    public RequestConfig requestConfigWithTimeout(int timeoutInMilliseconds) {
        return
RequestConfig.copy(RequestConfig.DEFAULT).setSocketTimeout(timeoutInMillisecond
s)

.setConnectTimeout(timeoutInMilliseconds).setConnectionRequestTimeout(timeoutIn
Milliseconds).build();
    }

    private void addBody(HttpRequestBase request, String body) {
        if(null != body && !"".equalsIgnoreCase(body)) {
            StringEntity entity = new StringEntity(body,
ContentType.APPLICATION_JSON);
            ((HttpEntityEnclosingRequestBase) request).setEntity(entity);
        }
    }

    private void addHeaders(HttpRequestBase request, Map<String, String>
headers) {
        if (headers != null) {
            for (Map.Entry<String, String> header : headers.entrySet()) {
                request.addHeader(header.getKey(), header.getValue());
            }
        }
    }


}
```

Additional Assignment

We encourage you to attempt the following optional assignment. While it is not mandatory, completing it may enhance your application and help strengthen your profile during our evaluation process.

Objective

Develop a Java-based scripting engine utility that dynamically executes **JavaScript** and **Python** scripts, returning their output as Java objects.

This should be done **entirely within the Java Virtual Machine (JVM)** without external command-line tools (e.g., `ProcessBuilder`). The solution should support the use of external libraries in both scripting languages.

Requirements

1. Create a **Java method** with the following signature (or similar):

```java
Object runScript(String language, String script);
```

2. The language parameter should support:

- "JavaScript"
- "python"

3. The script parameter should contain:

- Raw script content as a String
- (Optional enhancement: support for reading from a file)

4. The method must:

- Evaluate the script
- Return the result as a Java Object
- Support the use of libraries from JavaScript and Python

Constraints

- **Must not use ProcessBuilder or similar.**
- Can use any supported version of Java (Java 11 or later).
- Must allow importing and using standard and third-party libraries in both scripting languages(Example: Math, etc.).
- Optional: make the method thread-safe and reusable (e.g., via a service class).