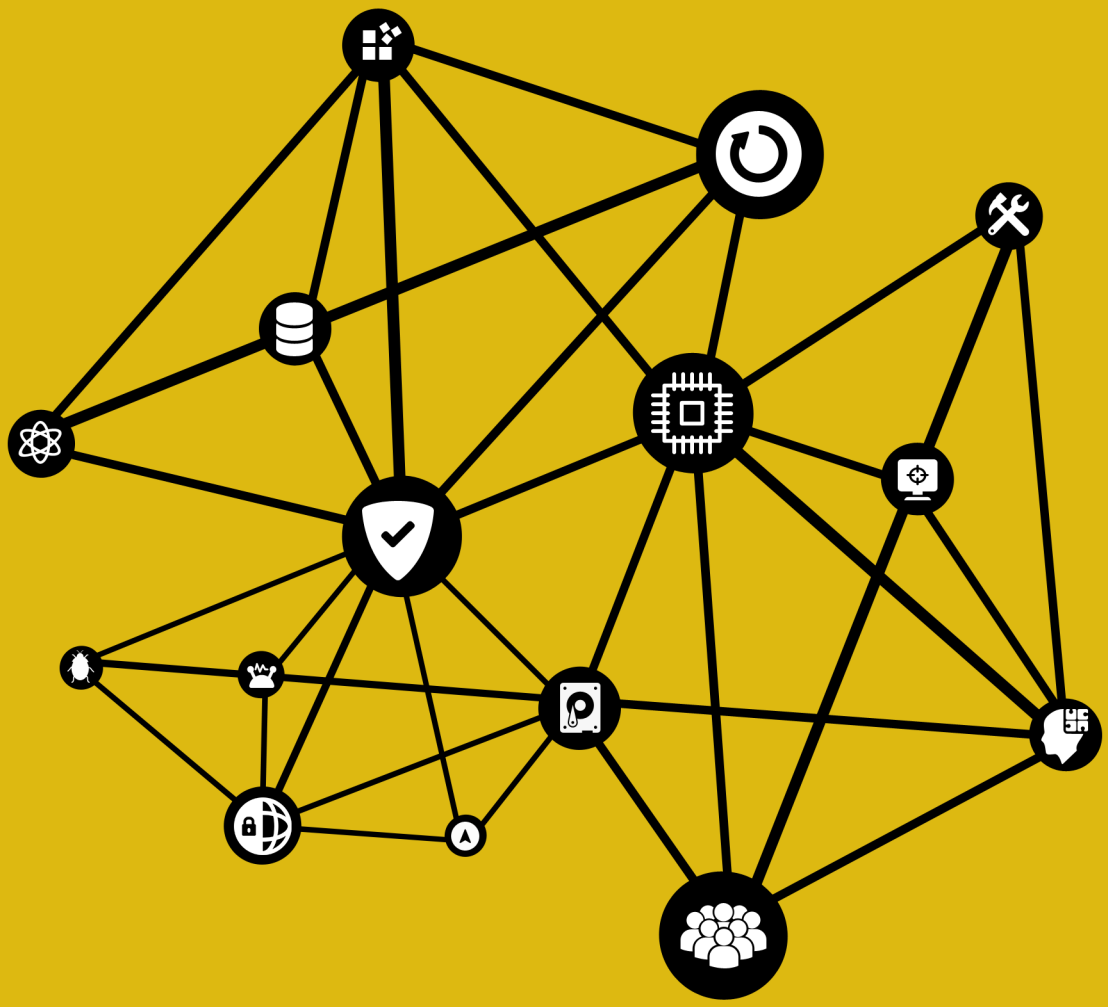
# BUILDING MICROSERVICES WITH SPRING BOOT



Engin Yöyen

# Building Microservices with Spring Boot

Engin Yöyen

This book is for sale at http://leanpub.com/building-microservices-with-spring-boot

This version was published on 2017-01-15

# Contents

# Preliminaries

## Errata & Suggestion

If you find any mistake in the text, code or in logic or you have a suggestion regarding the content, please do not hesitate to drop an email.

mail@enginyoyen.com

## Source Code

Examples that are giving in this book are freely available from the GitHub. Almost all of the code examples are in a git repository. You can access all of the source code from:

https://github.com/building-microservices-with-spring-boot[1]

Examples contains a partial url such as:

**Listing 1.3 - A first look at a Spring Boot - [git-repository]/ch01-microservices-and-spring/01**

```
1 ...
2 ...
3 ...
```

In this case, directory 01 will be in the following repository and the full URL would be:

https://github.com/building-microservices-with-spring-boot/ch01-microservices-and-spring[2]

## Credits

Social-media icon designed by Plainicon, from http://www.flaticon.com/authors/plainicon.

Bug, Network Protection, Plugin, Database, Locked Padlock, Navigation Arrow, CPU, Electrodes, Hard Drive, Energy Atom, Registry, Chip, Hacked Computer, energy, Tools Cross Settings Symbol for Interface, Crowd of Users, Reload Arrow, Web Guard icons designed by Freepik from http://www.flaticon.com/authors/freepik

[1] https://github.com/building-microservices-with-spring-boot

[2] https://github.com/building-microservices-with-spring-boot/ch01-microservices-and-spring

# 7. Communicating with HTTP Clients

Microservice style architecture is often build using REST over HTTP. Which also means, that services need to communicate with each other, hence information is distributed into smaller services. In the previous chapter, strategies for connecting HTTP APIs were introduced. In this chapter, I am going to use the basis of discussion from Chapter 6 and explain how HTTP services can communicate with each other by using Spring and Spring Boot.

## 7.1 Spring's Rest Template

`RestTemplate` is a class in spring web package, for synchronous client-side HTTP access. The sole aim is to simplify communication with HTTP servers. That is it actually this is all `RestTemplate` does. It offers methods that will handle HTTP operations very easily, such as retrieving data, posting data or retrieving all HTTP message headers.

### 7.1.1 Simple GET Request

Let's begin with a very simple example. We are going to request user information from GitHub. We are going to create a class and add the following properties as in Listing 7.1. `RestTemplate` will use this class to convert incoming JSON data into a `User` class which is defined in Listing 7.1. There are only three fields in the class, let's assume these are the only fields we are interested at the moment.

**Listing 7.1 - [git-repo]/ch07-communicating-with-http-clients/rest-template**

```
public class User {
    private String id;
    private String login;
    private String location;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getLogin() {
```

```
15          return login;
16      }
17
18      public void setLogin(String login) {
19          this.login = login;
20      }
21
22      public String getLocation() {
23          return location;
24      }
25
26      public void setLocation(String location) {
27          this.location = location;
28      }
29  }
```

The second step is to create a simple controller, which will get the username from the URI path and pass it to the `RestTemplate`. In this example, I am going to use `getForObject()` method retrieve a representation of a resource by doing a GET on the specified URL. The response is then converted to giving object and returned, in this case, the response will be `User` model which was created in Listing 7.1. Method accepts three parameters, URI template, class and URL variables. URI template variable has to be the same name of the method parameter, which you can see both are literal `username`. Otherwise, you will get an error.

**Listing 7.2 - [git-repo]/ch07-communicating-with-http-clients/rest-template**

```
1   @RestController
2   public class Controller {
3
4       RestTemplate restTemplate = new RestTemplate();
5
6       @RequestMapping(value = "/github/{username}")
7       User getUser(@PathVariable String username) {
8           User user = restTemplate
9                   .getForObject("https://api.github.com/users/{username}", User.cl\
10  ass, username);
11          return user;
12      }
13
14  }
```

When the code is compiled then you can test it via browser or `curl`, respond should look as in Listing 7.3. This is basically it, all you need to do for fetching something simple from an another

HTTP service. This is how you can connect another HTTP service to your Spring Boot application. As you may have noticed it, the code is very simple, but in real life, services may not be that reliable, so you need to add more effort to error handling, multiple requests, etc., which we will tackle in next section.

**Listing 7.3 - Service Response**

```
curl -i -X GET http://localhost:8080/github/enginyoyen

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{
        "id": "1239966",
        "login": "enginyoyen",
        "location": "Berlin, Germany"
}
```

## 7.1.2 Working with Response Entity

In the previous example information that is retrieved from the GitHub API is returned back to the user. However, this is not partially correct because some of the information such as HTTP Headers got lost. HTTP response contains also HTTP Headers, which are not transmitted to the client. Information such as HTTP Headers may not be useful to the client, or you may simply want to omit this information, in that case, the approach in the previous section will just do fine. However, if you need this information to be transmitted to the client, then there has to be a slide adjustment.

`getForObject` method returned the object, another method for retrieving the object is `getForEntity` which essentially does the same thing, but HTTP response is stored as a `ResponseEntity`. In some cases, you may actually return `ResponseEntity` directly as in the Listing 7.4.

**Listing 7.4 - Returning ResponseEntity directly**

```
1  @RestController
2  public class Controller {
3
4      RestTemplate restTemplate = new RestTemplate();
5
6      @RequestMapping(value = "/github/{username}")
7      ResponseEntity getUser(@PathVariable String username) {
8          ResponseEntity entity = restTemplate
9                  .getForEntity("https://api.github.com/users/{username}",
10                     User.class, username);
```

```
11          return entity;
12      }
13
14  }
```

This theoretically will work without any problem unless there has been a modification to the body of the response. GitHub's response is converted to the `User` object, but this object does not contain all the fields as in GitHub API. Controller responds only with partial information, that is returned from GitHub API. This is still a useful scenario, the problem is the `Content-Lenght` of the HTTP response from GitHub API, will not be the same Content-Length as of the Spring Boot application hence content is reduced to only three fields. When the code in Listing 7.4 is executed, the response will be as in Listing 7.5 as you can see all the HTTP Headers are present, but there is also an error message from `curl` (transfer closed with 1104 bytes remaining to read).

**Listing 7.5 - Service Response with Headers**

```
curl -i -X GET http://localhost:8080/github/enginyoyen
HTTP/1.1 200 OK
Server: GitHub.com
Date: Mon, 23 May 2016 11:27:34 GMT
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 40
X-RateLimit-Reset: 1464003944
Cache-Control: public, max-age=60, s-maxage=60
Vary: Accept-Encoding
ETag: "6da8a08a56a39efa07232d3f6868b5af"
Last-Modified: Wed, 27 Apr 2016 19:18:48 GMT
X-GitHub-Media-Type: github.v3
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X-Ra\
teLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X\
-Poll-Interval
Access-Control-Allow-Origin: *
Content-Security-Policy: default-src 'none'
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: d0b3c2c33a23690498aa8e70a435a259
X-GitHub-Request-Id: 5E45595B:B210:3FEF45A:5742E926
Content-Type: application/json;charset=utf-8
Content-Length: 1170
```

```
curl: (18) transfer closed with 1104 bytes remaining to read

{
    "id": "1239966",
    "login": "enginyoyen",
    "location": "Berlin, Germany"
}
```

To avoid this issue, simply removing `Content-Length` will be enough. Response header from the request is immutable, therefore, to alternate it must be copied. The code sample in Listing 7.6, creates a new `HttpHeaders` object and copies all the headers except `Content-Length` header and use this header to create a new response.

**Listing 7.6 - [git-repo]/ch07-communicating-with-http-clients/working-with-response-entity**

```
@RestController
public class Controller {

    RestTemplate restTemplate = new RestTemplate();

    @RequestMapping(value = "/github/{username}")
    ResponseEntity getUser(@PathVariable String username) {
        ResponseEntity entity = restTemplate
                .getForEntity("https://api.github.com/users/{username}",
                        User.class, username);

        HttpHeaders responseHeaders = new HttpHeaders();

        //Filter Content-Length header and add other to responseHeaders
        entity.getHeaders().entrySet()
            .forEach(
                header -> {
                    if (!header.getKey().equals(HttpHeaders.CONTENT_LENGTH)) {
                        header.getValue().forEach(
                            headerValue ->
                                responseHeaders.set(header.getKey(), headerValue)
                        );
                    }
                }
            );

        return new ResponseEntity(entity.getBody(),
            responseHeaders, HttpStatus.OK);
```

```
29      }
30  }
```

When you run the code again and make a HTTP request, you should now see that instead of `Content-Length` header there is a `Transfer-Encoding` header. This header added by the Spring, and means that data is will be transmitted in a series of *chunks*(see Chunked transfer encoding[3]). This simple example demonstrates how you can work or manipulate HTTP client response.

**Listing** 7.7

```
curl -i -X GET http://localhost:8080/github/enginyoyen
HTTP/1.1 200 OK
...
...
Transfer-Encoding: chunked

{
    "id": "1239966",
    "login": "enginyoyen",
    "location": "Berlin, Germany"
}
```

## 7.1.3 Other RestTemplate Methods

In the previous section, you have seen an example of how you can retrieve the data. The rules are exactly same for other operations. Therefore, there is no reason to go through every available method. In Table 7.1, you can see the overview of the `RestTemplate` methods that corresponds to HTTP method verbs. Method signatures are omitted because of the length.

**Table 7.1 - RestTemplate methods**

| Verb | Action | RestTemplate Method |
|---|---|---|
| GET | Responds with information about the resource | getForObject, getForEntity |
| POST | Creates a new resource | postForLocation, postForObject |
| PUT | Creates or updates the resource | put |
| DELETE | Deletes the resource | delete |
| HEAD | Gets all headers of the resource | headForHeaders |
| OPTIONS | Gets allow header of the resource | optionsForAllow |
| any | Execute the HTTP method to the resource | exchange, execute |

[3] https://en.wikipedia.org/wiki/Chunked_transfer_encoding

## 7.2 Handling Errors

Connecting HTTP services to each other brings another challenge, how do you handle errors? In monolith style software, it is a sequential process: handling errors are much more straight forward. While building microservice style architecture, you will be connecting many services to each other, there is no guarantee that services are available all the time neither response is not always successful. Therefore, it is important that each service response strictly with correct and meaningful way, meaning each service should either handle errors gracefully or make it transparent so the consumer of the service is aware of the problem. For instance, in the previous example, HTTP GET request to Spring Boot application returned response from GitHub. Let's try the same thing with a username that does not exist in GitHub as in Listing 7.4.

**Listing 7.8**

```
curl -i -X GET http://localhost:8080/github/user-that-does-not-exist

HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Sun, 15 May 2016 12:24:11 GMT
Connection: close

{
        "timestamp": 1463315051010,
        "status": 500,
        "error": "Internal Server Error",
        "exception": "org.springframework.web.client.HttpClientErrorException",
        "message": "404 Not Found",
        "path": "/github/user-that-does-not-exist"
}
```

Response from Spring Boot application is `500 Internal Server Error`, but if you try GitHub API directly, you will notice that it will respond with status code `404 Not Found`. This status code makes sense because the user does not exist. `RestTemplate` uses `DefaultResponseErrorHandler`, which naturally consider anything that is client error(4xx) or server error(5xx). However, it will help a client to give more or better information regarding the issue. Furthermore, sending correct HTTP status code with relevant information will help you to build better consumer service, hence, a consumer can depend on the response of the service. If the application returns only 500 HTTP status code for all errors, how does the consumer of that application should make any decision based on the response? The only reliable way for a consumer to take a correct decision based on service output is that output should be reliable.

There is a very easy solution to this problem, you can surround the `RestTemplate` with a `try-catch` block and handle the errors if any happens. Another alternative is to use `@ExceptionHandler` annotation to handle client exception. For more information about this annotation please refer to chapter *2.7 Handling Exceptions*.

The code in Listing 7.5 handles every `HttpClientErrorException` that occurs in the scope of controllers. Error handler method has a flexible signature, so for this example, the method returns `ResponseEntity` and accepts `HttpClientErrorException` class as a parameter. The method is a very simple one, it gets the response body, headers and status code of the HTTP request and returns it back as a response. To simply put, it pipes back the complete response(which considered error) from the client.

**Listing 7.9 - [git-repo]/ch07-communicating-with-http-clients/handling-errors**

```
@ControllerAdvice
class ExceptionHandlerAdvice {

    @ExceptionHandler(value = HttpClientErrorException.class)
    ResponseEntity<?> handleHttpClientException
        (HttpClientErrorException exception) throws Exception
    {
        return new ResponseEntity(exception.getResponseBodyAsByteArray(),
                exception.getResponseHeaders(),
                exception.getStatusCode());
    }
}
```

When you compile and run the application, HTTP request(with existing username) to the application will return the same result as in Listing 7.6. However, when GitHub API generates HTTP response in the range of what is considered error such as 4xx or 5xx(for instance request with the wrong username), application pipes back the response of the client as in Listing 7.10.

**Listing 7.10**

```
curl -i -X GET http://localhost:8080/github/user-that-does-not-exist

HTTP/1.1 404 Not Found
Server: GitHub.com
Date: Wed, 18 May 2016 19:43:07 GMT
Status: 404 Not Found
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 59
X-RateLimit-Reset: 1463604187
X-GitHub-Media-Type: github.v3
```

```
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X-Ra\
teLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X\
-Poll-Interval
Access-Control-Allow-Origin: *
Content-Security-Policy: default-src 'none'
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-GitHub-Request-Id: 5F5BF61E:B215:93E3337:573CC5CA
Content-Type: application/json;charset=utf-8
Content-Length: 77

{
    "message": "Not Found",
    "documentation_url": "https://developer.github.com/v3"
}
```

This approach is helpful because there might be other errors that occur, for instance, GitHub applies rate limiting, and if you are making a unauthenticated request, you can make up to 60 requests per hour. When that number is exceeded server response with `403 Forbidden` status code. With current error handling mechanism, this message is completely transparent, see Listing 7.11.

**Listing 7.11**

```
curl -i -X GET http://localhost:8080/github/user-that-does-not-exist

HTTP/1.1 403 Forbidden
Server: GitHub.com
Status: 403 Forbidden
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1463604187
Content-Type: application/json;charset=utf-8
Content-Length: 246

{
    "message": "API rate limit exceeded for xxx.xxx.xxx.xxx.
        (But here's the good news: Authenticated requests get a higher
        rate limit. Check out the documentation for more details.)",
    "documentation_url": "https://developer.github.com/v3/#rate-limiting"
}
```

# 7.3 Asynchronous HTTP Requests

Microservice style system can be a very chatty system. In monolithic style software, information retrieval is simply a method call in the same process(at least in most of them). As the system gets divided into smaller services, naturally services start to communicate with each other. Previous method calls in the same process, are now HTTP calls into different isolated services. Service A depends on Service B and C, Service B depends on Service D and that depends on datastore, etc. Once you have all the components in place, you need to ensure the performance of the system. As you would remember, in previous Chapter *6.3 API Gateway*, there was an example of a client making a single request and service merging all the request from different services.

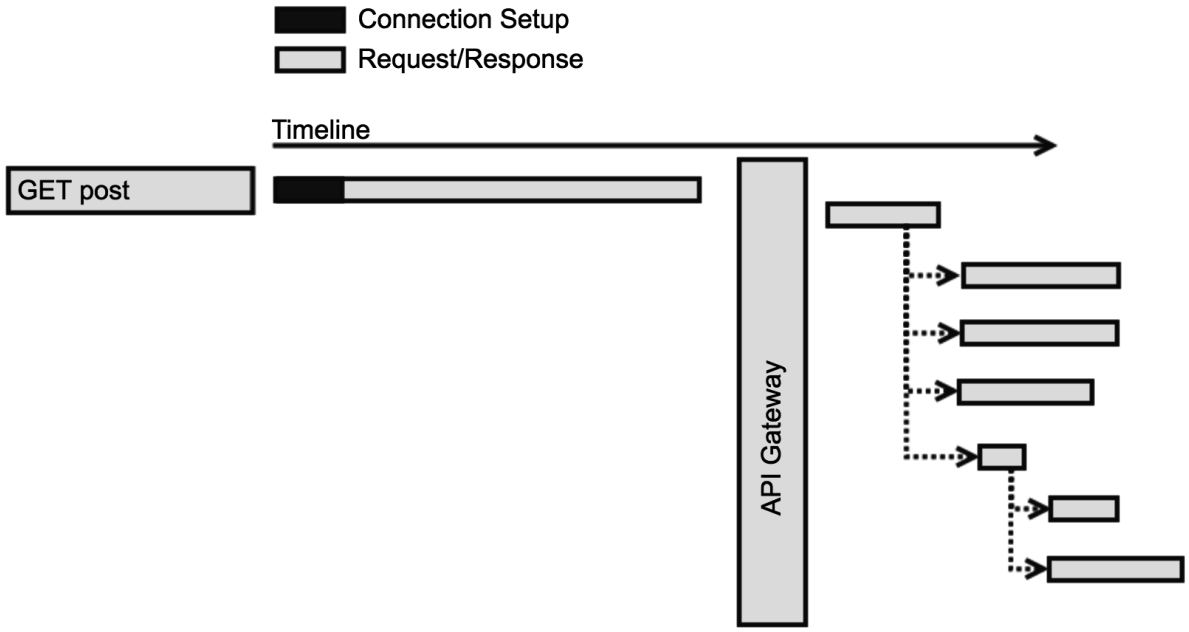



**Figure 7.1 - Merging Request in Service**

If we take that example again, in a sequential program, each HTTP request has to be executed and has to be completed either with a succesful result or a failure, for next HTTP call to be made, hence method cannot proceed forward. Math is very clear, if there are *n* number of HTTP requests, you have to add up request-response time of each request, and local processing time of retrieved data, which will be the total time needed for an application to respond a single HTTP request.

An alternative to this synchronous approach is creating asynchronous tasks that will execute in background parallel. So you will reduce the request-response time to longest execution time of a background task and local processing time. Usually, most developers are happy when they do not have to get they hands dirty with threads, but in this case, the task is relatively simple, so no need

to worry about it.

To demonstrate the functionality, I will use the same GitHub example in section 7.1.1 and extend it. URI will stay same, but Spring Boot application will respond with:

- user information (same as in section 7.1.1) of the given username
- name of all the repositories of the given username
- name of all the organizations of the given username

There will be three separate asynchronous HTTP request for each information group and when all of them are retrieved it will be merged and delivered to the client as a response.

First we need to create the data model for repositories and organizations, I will pick up only one field, which should be enough to demonstrate the functionality.

**Listing 7.12 - Repositories - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
1  public class Repositories {
2      private String name;
3
4      public String getName() {
5          return name;
6      }
7
8      public void setName(String name) {
9          this.name = name;
10     }
11 }
```

**Listing 7.13 - Organizations - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
1  public class Organizations {
2      private String login;
3
4      public String getLogin() {
5          return login;
6      }
7
8      public void setLogin(String login) {
9          this.login = login;
10     }
11 }
```

All three requests that are made to GitHub API are separate but the application will merge and response back to a single HTTP request. Therefore, I will extend the `User` class to add repositories and organizations.

**Listing 7.14 - Organizations - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
public class User {
    private String id;
    private String login;
    private String location;


    private List<Repositories> repositories;

    private List<Organizations> organizations;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public List<Repositories> getRepositories() {
        return repositories;
    }

    public void setRepositories(List<Repositories> repositories) {
        this.repositories = repositories;
    }
```

```
42
43      public List<Organizations> getOrganizations() {
44          return organizations;
45      }
46
47      public void setOrganizations(List<Organizations> organizations) {
48          this.organizations = organizations;
49      }
50  }
```

I will create a class called `GitHubClient` and so I can add all the HTTP request methods in this class. This class will be annotated with `@Service` annotation so I can wire up in the controller.

**Listing 7.15 - GitHubClient - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
1  @Service
2  public class GitHubClient {
3      RestTemplate restTemplate = new RestTemplate();
4
5  }
```

Next thing we need to do, is to create the method that makes HTTP request, we just have to use the same code as before, which will be as in Listing 7.16.

**Listing 7.16 - GitHubClient - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
1  public User getUser(@PathVariable String username)
2      {
3      User user = restTemplate
4              .getForObject("https://api.github.com/users/{username}",
5                  User.class, username);
6
7      return user;
8  }
```

To make that HTTP request method asynchronously, there are two changes, we need to do for that, first, we need to mark the method with `@Async` annotation, so Spring will run that in a separate thread. Second we need to return the response type of `CompletableFuture<T>` instead of `User`. `CompletableFuture` represent the result of an asynchronous computation that may be explicitly completed. This class is part of JDK 1.8(see CompletableFuture[4]). You can see the code in Listing 7.17, there has been no adjustment to HTTP request and when it is completed, it will

[4] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

return `CompletableFuture` an already completed value. Furthermore, you can see that before values are returned, the thread is paused for 2 seconds, this is only added to artificially produce a delay, so we can verify the functionality.

**Listing 7.17 - GitHubClient - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
@Async
public CompletableFuture<User> getUser(@PathVariable String username)
    throws Exception
    {
    User user = restTemplate
            .getForObject("https://api.github.com/users/{username}",
                User.class, username);
    Thread.sleep(2000L);
    return CompletableFuture.completedFuture(user);
}
```

Retrieving organizations and repositories are in equally same, therefore, there is no need to explain the code. You can see the complete class in Listing 7.18.

**Listing 7.18 - Complete GitHubClient - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
@Service
public class GitHubClient {

    RestTemplate restTemplate = new RestTemplate();

    @Async
    public CompletableFuture<User> getUser(@PathVariable String username)
        throws Exception
        {
        User user = restTemplate
                .getForObject("https://api.github.com/users/{username}",
                    User.class, username);
        Thread.sleep(2000L);
        return CompletableFuture.completedFuture(user);
    }

    @Async
    public CompletableFuture<List<Repositories>> getRepos
        (@PathVariable String username) throws Exception
        {
        Repositories[] user = restTemplate
```

```
22                  .getForObject("https://api.github.com/users/{username}/repos",
23                      Repositories[].class, username);
24          Thread.sleep(2000L);
25          return CompletableFuture.completedFuture(Arrays.asList(user));
26      }
27
28
29      @Async
30      public CompletableFuture<List<Organizations>> getOrganizations
31          (@PathVariable String username) throws Exception
32          {
33          Organizations[] user = restTemplate
34                  .getForObject("https://api.github.com/users/{username}/orgs",
35                      Organizations[].class, username);
36          Thread.sleep(2000L);
37          return CompletableFuture.completedFuture(Arrays.asList(user));
38      }
39  }
```

In Listing 7.19 you can see the controller that now uses the `GitHubClient` class to make HTTP requests. Methods `getUser(String username)`, `getRepos(String username)` and `getOrganizations(String username)` are called one after another without waiting for HTTP request to be completed. After all three HTTP requests are initiated, static `allOf` method is being called with all three `CompletableFutures`. What this method does is it takes an array of futures and returns a `CompletableFutures` that is completed when all of the given `CompletableFutures` are completed. The `join()` method afterward, returns the result value when a future is completed. Long story short: `allOf` and `join` awaits completion of a set of independent CompletableFutures before continuing with the execution of the code. This is beneficial, hence, we need to return all of the information that is retrieved at once. Next line simply references the `User` object for ease of readiablity, `get()` method on future waits if necessary to complete, and then returns its result, but the previous line already made sure of that. Organizations and repositories are assigned to `User` object and this is returned back. `Stopwatch`(from google's guava library) at the beginning and at the end of the method is used to give more information about the length of the processing time.

**Listing 7.19 - Complete Controller - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
@RestController
public class Controller {

    @Autowired
    GitHubClient gitHubClient;

    Logger logger = LoggerFactory.getLogger(Controller.class);

    @RequestMapping(value = "/github/{username}")
    User getUser(@PathVariable String username) throws Exception {
        Stopwatch stopwatch = Stopwatch.createStarted();

        CompletableFuture<User> maybeUser = gitHubClient.getUser(username);
        CompletableFuture<List<Repositories>> maybeRepos  =
            gitHubClient.getRepos(username);
        CompletableFuture<List<Organizations>> maybeOrgs  =
            gitHubClient.getOrganizations(username);

        CompletableFuture.allOf(maybeUser, maybeRepos, maybeOrgs).join();

        User user = maybeUser.get();

        user.setRepositories(maybeRepos.get());
        user.setOrganizations(maybeOrgs.get());

        stopwatch.stop();
        logger.info("All request completed in " +
            stopwatch.elapsed(TimeUnit.MILLISECONDS));

        return user;
    }
}
```

If you go ahead and give it a try response of the service should be above 6 seconds as each request is being paused 2 seconds(see Listing 7.17 and 7.18). Which I urge you to try. Only missing part is to tell Spring container asynchronous method execution capability, to do you need to add `@EnableAsync` annotation. After adding this annotation you can test it again and now response you get should be a bit above 2 seconds.

**Listing 7.20 - Complete Controller - [git-repo]/ch07-communicating-with-http-clients/asynchronous-http-requests**

```
@EnableAsync
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

When you execute the code you should see the repositories and organizations are delivered as well as in Listing 7.21.

**Listing 7.21**

```
curl -i -X GET http://localhost:8080/github/enginyoyen

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

{
    "id": "1239966",
    "login": "enginyoyen",
    "location": "Berlin, Germany"
    "repositories":[
        ...
    ],
    "organizations":[
        ...
    ]
}
```

This is a very simple example, but it shows you that with bit more work you can actually implement asynchronous tasks and for sure it will make a lot of improvements to the performance of your system. If you are connecting many clients and doing multiple operations you need to take advantage of concurrency unless you have a very good reason not to do so.

## 7.4 Handling Domain Model Across Services

Each microservice delivers a sort of result representing the business domain. The assumption of course that each service is divided well enough that they do not need to know the internals of the

other services. But then again, there is always an intersection in a system where services have to communicate with each other, which also means they have to understand what the other service offer as a domain model. This may sound a very regular issue and indeed, it is, but before we dive into solution context, let's examine a use case. In Figure 7.2, a small part of a call-center system setup is shown. The user of this system are usually companies who build help-desks, hotlines, etc. So software is used by so called agents, who answers or calls some customer and by admins who plans to shifts, adjust groups, monitors calls etc.

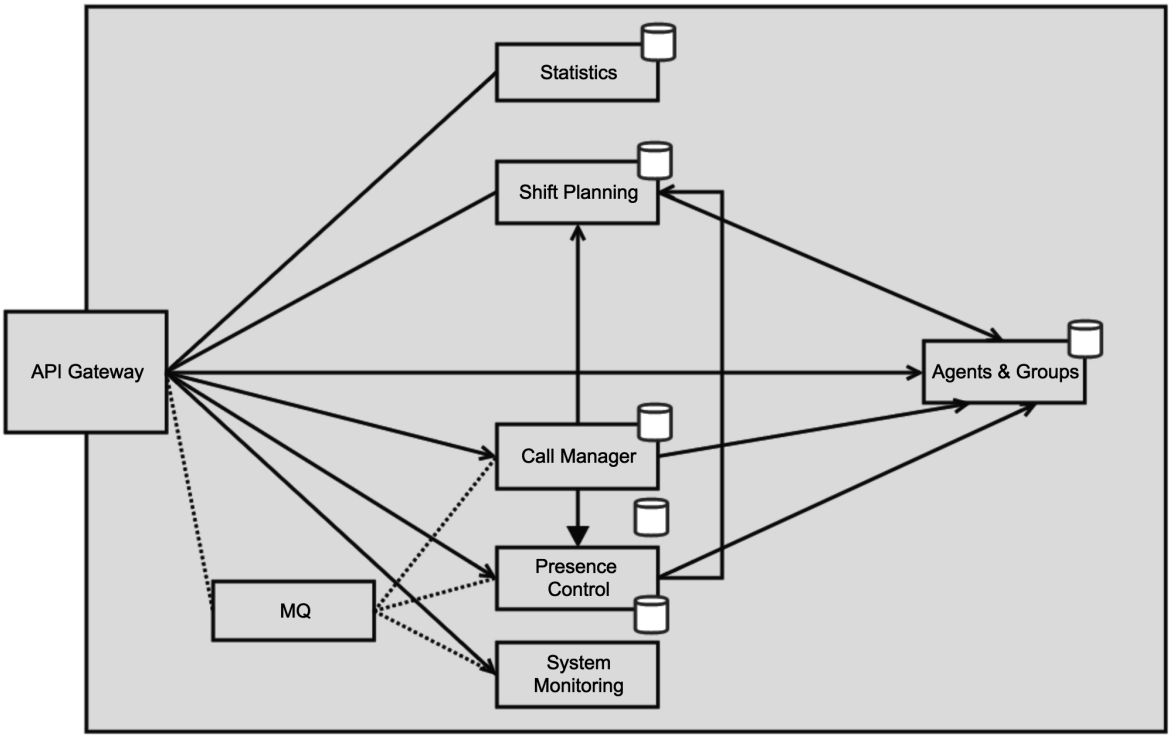



**Figure 7.2 - Merging Request in Service**

Each square represents a service, all straight lines uses HTTP protocol whereas dashed lines are AMQP. The dependency of the system is shown with the direction of the arrow. Boxes that has database icon connects to the database.

- **API Gateway** : Handles authentication, some part of authorization, acts as a proxy to other services
- **Statistics** : Gathers and delivers all system statistics
- **Shift Planning** : Handles all shift planning, meaning system response which agent is available at what time and date. Also used by agents to add their work times.
- **Agent & Groups** : Delivers agent information and which call group(order,refunds, payments,etc.) agent is working in
- **Call Manager** : Initiated and routes incoming calls
- **Presence Manager** : Handles the presence of agents(e.g. available, on call, on break, etc.)

- **System Monitoring** : Checks system and call conditions, triggers system alerts, reacts on call peaks

Now, as you can some components depends on *Agents&Groups* service, *Call-Manager* depends on *Presence Control* and *Shift Planning* to know which agent should be working and which agent is available so it can route to calls. Furthermore, three of the components connects to the Message Queue to push information changes to a client immediately. Assuming each service has its own domain model and business logic how do they share core entities across services?

### 7.4.1 DRY : Code Reuse by Packaging Domain Model as Jar file

This is an answer that will just pop up to everyone minds. It is great, you do not have to rewrite the domain model in every service. It is software development 101 : *Don't repeat yourself (DRY)*(see DRY[5]). But this let's overly coupled services, where the goal was to decouple all services!

For instance, let's assume we have added a new field in *Agents&Groups* service and build process produced a new jar file with a new version and added into the artifact repository. Now we have to increase the library version in all the applications using this domain model. It is a very dummy work, so we can configure the build process to do it automatically. Now the result is, every time I make a minor change (e.g. delete a comment), all services have to update the version of the library, which also means that there is a new version of the services. This in-turn triggers a new round of updates. After a while, it is also not transparent why you have a 100 new version of a single service in 1 month!. Chance logs and commits are good as useless, hence, there is no transparency.

### 7.4.2 WET : Write Everything Twice (or more)

A lot of repetition(maybe), but gives you the flexibility of decoupled services. In an ideal world, information that is being exchanged between different services should be minimum. Each service should have its own context and business logic. Other services can act as a consumer and use information that is provided, but it should not need to complete domain model. Getting back to the example in Figure 7.2, *Call Manager* needs to know, whether an agent has a shift or not, but does not need to know, how shifts are scheduled, which days there are free slots, or which time format it is used to persist in the database. Most of the information that *Shift Planning* handling is irrelevant, therefore code duplication does not hurt.

Long story short, unless you MUST, do not try to share the code between your services, it will only increase your pain in the long term.

### 7.4.3 Resources

Eric Evan's book *Domain-Driven Design: Tackling Complexity in the Heart of Software*, it is a good resource to read on regarding domain-design. One of the ideas in his book called, Bounded Context,

[5] https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

which means any domain consist of multiple bounded contexts, each context has multiple models, but there is no need to communicate these models to outside world. I suggest that you read the book, but if you need quick introduction, please check out following links:

*Eric Evans on Domain-Driven Design at 10 Years*: http://www.se-radio.net/2015/05/se-radio-episode-226-eric-evans-on-domain-driven-design-at-10-years/[6]

*Bounded Context*: http://martinfowler.com/bliki/BoundedContext.html[7]

# 7.5 Netflix's Zuul

Zuul is an open source project from Netflix, which is used in Netflix for performing different sort of HTTP operations such as authentication, dynamic routing, monitoring, load shedding, etc. Zuul depends heavily on filters, each filter performs some action on each HTTP request/response. So the combination of the filters gives a control over entire service. This filtering approach gives a flexibility which can be used in many different ways. The aspect that I am going to focus in this section is routing and how filters are working.

## 7.5.1 Routing

Routing(in this case) simply refers to forwarding client traffic to particular service. In Chapter 6.1 An Example Scenario was giving to introduce the structure how traffic can be forwarded to multiple services. The approach is valid but if you need some extra processing for each HTTP request (e.g. authentication, caching, rate-limiting) using pure HTTP reverse proxy will be very challenging. On the other hand, if you decide to use API Gateway approach, using `RestTemplate` can do the job as in section 7.1.1, but it is quite a lot of work, every time you have a new HTTP endpoint, you need to define a similar endpoint in API Gateway as well. Especially if you expose many different URI endpoints to clients. As an alternative, you can use Netflix's Zuul. Zuul filters can perform required authentication, debugging, routing and will enable you to add new HTTP endpoint to the API Gateway is just a matter of couple lines of configuration. The following example will demonstrate the same functionality as in section 7.1.1 and 7.1.2 with Zuul.

To begin with, we need to create a new Spring Boot application and add `spring-cloud-starter-zuul` dependency as in Listing 7.22. If you need help to manage your POM file with multiple parents please refer Appendix A.1.

[6] http://www.se-radio.net/2015/05/se-radio-episode-226-eric-evans-on-domain-driven-design-at-10-years/
[7] http://martinfowler.com/bliki/BoundedContext.html

**Listing 7.22 - [git-repo]/ch07-communicating-with-http-clients/spring-cloud-zuul**

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Camden.SR4</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>

    ...
<dependencies>
```

Next, we need to enable Zuul by annotating the main class with `@EnableZuulProxy` annotation.

**Listing 7.23 - [git-repo]/ch07-communicating-with-http-clients/spring-cloud-zuul**

```
1  @EnableZuulProxy
2  @SpringBootApplication
3  public class Application {
4      public static void main(String[] args) {
5          SpringApplication.run(Application.class, args);
6      }
7  }
```

Now at this stage, we have added the dependency and enabled Zuul. Remaining work is to define the where requests will be forwarded. To do that, we need to specify the location of the client(as a URL as service discovery is not in place) and the path of the local application, as you can see in Listing 7.24. The current setup means that any calls `/github/**` will be forwarded to `https://api.github.com/users`.

**Listing 7.24 - [git-repo]/ch07-communicating-with-http-clients/spring-cloud-zuul**

```
zuul.routes.github.path: /github/**
zuul.routes.github.url: https://api.github.com/users
zuul.routes.github.serviceId: github
```

When you compile and run the application you can test the functionality of routing by making a GET request and the response should look as in Listing 7.25(content details are omitted because of the length). Response from Spring Boot application will return complete content and HTTP headers without any addtional effort.

**Listing 7.25**

```
curl -i -X GET http://localhost:8080/github/enginyoyen

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Application-Context: bootstrap
....


{
  "login": "enginyoyen",
  "id": 1239966,
  "avatar_url": "https://avatars.githubusercontent.com/u/1239966?v=3",
  ...
}
```

## 7.5.2 Filters

Filters are designed to operate on the lifecycle of the HTTP requests and responses. Each filter can perform different type of action on different stage of HTTP request or response, therefore, there are 4 different types of filters:

- **pre**: Filters that are executed before the request is routed (e.g. authentication, logging)
- **routing**: Filters that can handle the actual routing of the request,
- **post**: Filters that are executed after the request has been routed
- **error**: Filters that are executed on HTTP request lifecycle

First three filters, will run for every request(if `shouldFilter()` method returns `true` indicating that the filter should run), while the last one only when an error occurs. There is no communication

between filters, however, they can share they state through a `RequestContext` which is unique to each request.

Creating custom filter is very simple, all filters must extend `ZuulFilter` class and implement the following four methods

- `String filterType()` : Returns type of filter(e.g. pre, routing, post, error)
- `int filterOrder()` : Returns order of the filter
- `boolean shouldFilter()` : Returns boolean value indicating if filter should be applied to that particular request
- `Object run()` : This method contains actual functionality of the filter

To give an example, I will create a two filter type of `pre` and `post` called `DebugFilter` and `ResponseHeaderFilter`. As names suggest, first filter just prints out basic HTTP request information, while the second one adds a simple header to HTTP response message. The filter can be seen in Listing 7.26 and 7.27. Both filter returns `true` for `shouldFilter()` method which means it will run for every HTTP request and response.

**Listing 7.26 - [git-repo]/ch07-communicating-with-http-clients/spring-cloud-zuul-filters**

```
1  public class DebugFilter extends ZuulFilter {
2
3      private static Logger logger = LoggerFactory.getLogger(DebugFilter.class);
4
5
6      public DebugFilter() {
7      }
8
9      public String filterType() {
10         return "pre";
11     }
12
13     public int filterOrder() {
14         return 10000;
15     }
16
17     public boolean shouldFilter() {
18         return true;
19     }
20
21     public Object run() {
22         RequestContext ctx = RequestContext.getCurrentContext();
23         HttpServletRequest request = ctx.getRequest();
```

```
24          logger.info(String.format("%s,%s,%s", request.getMethod(),
25              request.getRequestURI(), request.getProtocol()));
26          return null;
27      }
28  }
```

**Listing 7.27 - [git-repo]/ch07-communicating-with-http-clients/spring-cloud-zuul-filters**

```
1   public class ResponseHeaderFilter extends ZuulFilter {
2
3       public ResponseHeaderFilter() {
4       }
5
6       public String filterType() {
7           return "post";
8       }
9
10      public int filterOrder() {
11          return 100;
12      }
13
14      public boolean shouldFilter() {
15          return true;
16      }
17
18
19      public Object run() {
20          HttpServletResponse servletResponse = RequestContext
21              .getCurrentContext().getResponse();
22          servletResponse.addHeader("X-Spring-Boot-Proxy", "Zuul");
23          return null;
24      }
25
26  }
```

Once classes are created, remaining work is to register as classes as beans to Spring's container with `@Bean` annotation as you can see in Listing 7.28.

**Listing 7.28 - [git-repo]/ch07-communicating-with-http-clients/spring-cloud-zuul-filters**

```
@EnableZuulProxy
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public DebugFilter debugFilter() {
        return new DebugFilter();
    }

    @Bean
    public ResponseHeaderFilter addHeaderFilter() {
        return new ResponseHeaderFilter();
    }
}
```

When you compile, run the application and make an HTTP request, you can see the log message from the console and from HTTP response, you can see the newly added HTTP header as in Listing 7.29.

**Listing 7.29**

```
curl -i -X GET http://localhost:8080/github/enginyoyen

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Application-Context: bootstrap
X-Spring-Boot-Proxy: Zuul
....


{
  "login": "enginyoyen",
  "id": 1239966,
  "avatar_url": "https://avatars.githubusercontent.com/u/1239966?v=3",
  ...
}
```

Spring Cloud contains Zuul filters that are enabled by default. You can see these filters in `org.springframework.cloud.netflix.zuul.filters` package or from source code repository:

- https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters [8]

Another additional resource is Zuul source code repository:

- https://github.com/Netflix/zuul/tree/1.x/zuul-simple-webapp/src/main/groovy/filters[9]

## 7.6 HTTP Connection Pool

Communication between microservices is not process call, which means there is communication overhead between services. In short term, it is an additional network latency for each piece of data that is required. Establishing communication between one host to another involves multiple package exchanges between services, which is a resource consuming process. There is an alternative, however, instead of establishing a new connection on every HTTP request, it is possible to reuse open connections by executing multiple requests. The ability to use a single TCP connection to send and receive multiple HTTP requests and responses is called *HTTP connection persistence* or *HTTP keep-alive*. Figure 7.3 outlines this idea very clearly.

[8] https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters

[9] https://github.com/Netflix/zuul/tree/1.x/zuul-simple-webapp/src/main/groovy/filters

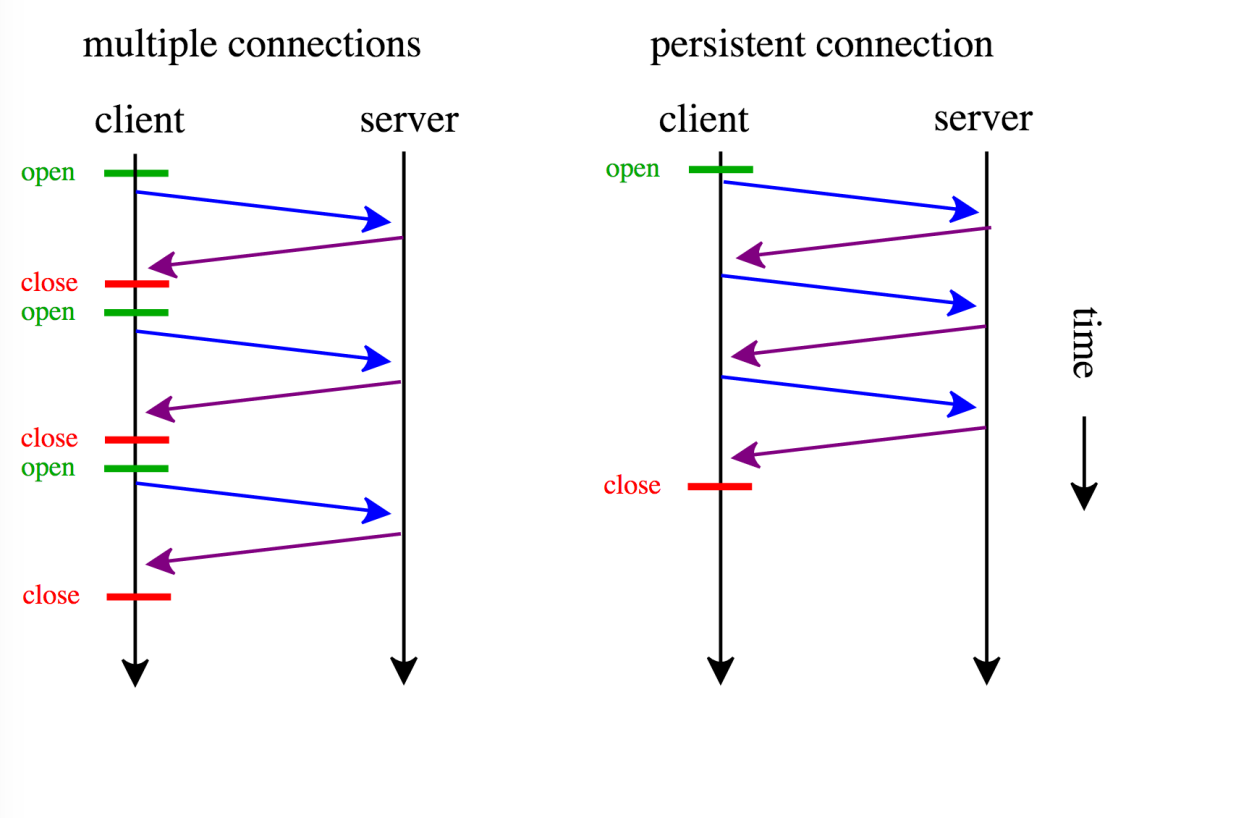



**Figure 7.3 - HTTP Connection Persistence**

*Source : https://en.wikipedia.org/wiki/HTTP_persistent_connection#/media/File:HTTP_persistent_connection.svg*[10]

As most services are multi-threaded, it is a useful strategy to have a pool of HTTP connections which has established connections to the target host(s). The job of the connection pool is to create a new connection when it is necessary or lease an existing connection, validate connection and manage the pool. By doing this, a multi-threaded application can lease an open connection from the pool and release it when its response of the request is delivered.

`RestTemplate` by default does not use any HTTP connection persistence. Furthermore, it uses standard JDK facilities to establish HTTP connections. However, it is possible to use a different HTTP client libraries with `RestTemplate`, such as Apach HttpClient[11] or OkHttp[12]. In this section, I am going to use Apache HttpClient to configure and use a connection pool within a Spring Boot application.

[10] https://en.wikipedia.org/wiki/HTTP_persistent_connection#/media/File:HTTP_persistent_connection.svg
[11] http://hc.apache.org/
[12] http://square.github.io/okhttp/

## 7.6.1 Configuring Apache HttpClient

To give an example how the connection pool is working I will use the same example as in the beginning of this chapter, that is simply making an HTTP request to GitHub HTTP API with the provided username and returning a user profile. The only difference will be, that this time we are going to use the HTTP connection pool instead of creating new HTTP connection on every request.

To begin with, you need to add Apache HttpClient dependencies to your POM file.

**Listing 7.30 - Apache HttpClient Dependency - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.2</version>
</dependency>
```

Once the library is available in the classpath, we can start by adding some connection pool configuration to the properties file. Externalizing those properties will allow you to alter application behavior without a new deployment.

**Listing 7.31 - Configuration Properties - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
httpClient.clientPoolSize = 10
httpClient.defaultMaxPerRoute = 10
httpClient.validateAfterInactivity = 3000
httpClient.connectionRequestTimeout = 5000
httpClient.connectTimeout = 5000
httpClient.socketTimeout = 5000

logging.level.org.apache.http.impl.conn = DEBUG
```

- *clientPoolSize* : Maximum number of connections in the pool.
- *defaultMaxPerRoute* : Maximum number of connection per route. In this example, we have a single route, therefore, the number is same as the max total.
- *validateAfterInactivity* : Period of inactivity in milliseconds after which persistent connections must be re-validated prior to being leased.
- *connectionRequestTimeout* : Timeout in milliseconds used when requesting a connection from the connection manager.
- *connectTimeout* : Timeout in milliseconds until a connection is established.
- *socketTimeout* : Timeout in milliseconds, which is the maximum period of inactivity between two consecutive data packages.

And the last entry in the properties file sets log level to DEBUG, this should allow us to see the logs of the connection pool manager.

To configure the connection pool, I will create a class called `HttpConnectionPoolConfig` and annotate with `@Configuration` annotation. This class will initialize the connection pool and the `RestTemplate`. We begin by adding variables that are going to be used in the configuration of the connection pool. If you are not sure how to load externalized configuration, you can refer *Chapter 4 - Application Configuration* for details.

**Listing 7.32 - HttpConnectionPoolConfig class - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
@Configuration
public class HttpConnectionPoolConfig {

    @Value("${httpClient.clientPoolSize}")
    private int clientPoolSize;

    @Value("${httpClient.defaultMaxPerRoute}")
    private int defaultMaxPerRoute;

    @Value("${httpClient.validateAfterInactivity}")
    private int validateAfterInactivity;

    @Value("${httpClient.connectionRequestTimeout}")
    private int connectionRequestTimeout;

    @Value("${httpClient.connectTimeout}")
    private int connectTimeout;

    @Value("${httpClient.socketTimeout}")
    private int socketTimeout;
}
```

Once the class is there with all variables in it, we can start adding the first method in the class that will create an instance of a `PoolingHttpClientConnectionManager`, which manages a pool of client connections and we annotate with `@Bean` annotation. Here we set up pool size, max connection per route and when should connections should be re-validated prior to being leased.

**Listing 7.33 - Connection Manager Configuration - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
@Bean
public PoolingHttpClientConnectionManager initConnectionManager() {
    PoolingHttpClientConnectionManager poolManager
        = new PoolingHttpClientConnectionManager();
    poolManager.setMaxTotal(clientPoolSize);
    poolManager.setDefaultMaxPerRoute(defaultMaxPerRoute);
    poolManager.setValidateAfterInactivity(validateAfterInactivity);
    return poolManager;
}
```

Going forward, we initialise `CloseableHttpClient`, in the same class, which is to simply put a closable HTTP client, hence the name. This method accepts `PoolingHttpClientConnectionManager` as a parameter, which Spring will use the one that has been initiated. Method contains additional `SocketConfig` and `RequestConfig` which are used to initiate `CloseableHttpClient` and return it.

**Listing 7.34 - HttpClient Configuration - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
@Bean
public CloseableHttpClient initHttpClient
        (PoolingHttpClientConnectionManager connectionManager) {

    SocketConfig socketConfig = SocketConfig.custom()
            .setSoKeepAlive(true)
            .setSoReuseAddress(true)
            .setTcpNoDelay(true)
            .build();

    RequestConfig requestConfig = RequestConfig.custom()
            .setConnectionRequestTimeout(connectionRequestTimeout)
            .setConnectTimeout(socketTimeout)
            .setSocketTimeout(connectTimeout)
            .build();

    CloseableHttpClient result = HttpClientBuilder
            .create()
            .setConnectionManager(connectionManager)
            .setDefaultSocketConfig(socketConfig)
            .setDefaultRequestConfig(requestConfig)
            .build();
    return result;
}
```

The final bean that we need to create is the `RestTemplate` bean. To instantiate the `RestTemplate` we need to create a new `HttpComponentsClientHttpRequestFactory` which uses Apache HttpClient to create HTTP requests, and we pass that request factory object to `RestTemplate`. `RestTemplate` will then use Apache HttpClient to make all the request and therefore take advantage of the HTTP connection pool.

**Listing 7.35 - RestTemplate Configuration - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
@Bean
public RestTemplate initRestTemplate(HttpClient httpClient) {
    HttpComponentsClientHttpRequestFactory requestFactory =
        new HttpComponentsClientHttpRequestFactory();
    requestFactory.setHttpClient(httpClient);
    return new RestTemplate(requestFactory);
}
```

## Exposing Objects as Bean

It is well possible to initialize all the objects in the `initRestTemplate()` method of `HttpConnectionPoolConfig` and return simply the `RestTemplate`. However, in upcoming chapters(Chapter 12, Section 12.3.2 Exposing Application Metrics on Endpoint Request) I will be using this example to expose application metrics, this is the reason why `PoolingHttpClientConnectionManager` is exposed as a separate bean.

The only remaining task is to inject `RestTemplate`, instead of creating an instance of it in the controller. The different is, when you inject the `RestTemplate`, Spring will use the one that was created in the Listing 7.35. If you created a new one(e.g. `RestTemplate restTemplate = new RestTemplate()`), the new instance of the object will not be taking advantage of the connection pool. The controller will look as follow;

**Listing 7.36 - Injecting RestTemplate - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
@RestController
public class Controller {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/github/{username}")
    User getUser(@PathVariable String username) {
        User user = restTemplate
                .getForObject("https://api.github.com/users/{username}",
```

```
11                  User.class, username);
12          return user;
13      }
14
15  }
```

Once you start the application and start making concurrent requests, you can see the console output and see that HTTP connections are begin created, leased and released as in Listing 7.37.

**Listing 7.37 - Console Output - [git-repo]/ch07-communicating-with-http-clients/http-connection-pool**

```
*** : Connection request: [route: {s}->https://api.github.com:443]
    [total kept alive: 0; route allocated: 0 of 10; total allocated: 0 of 10]
*** : Connection leased: [id: 0][route: {s}->https://api.github.com:443]
    [total kept alive: 0; route allocated: 1 of 10; total allocated: 1 of 10]
*** : Connecting to api.github.com/192.30.253.117:443
*** : Connection established 192.168.2.108:57802<->192.30.253.117:443
*** : http-outgoing-0: set socket timeout to 5000
*** : Connection [id: 0][route: {s}->https://api.github.com:443]
    can be kept alive indefinitely
*** : Connection released: [id: 0][route: {s}->https://api.github.com:443]
    [total kept alive: 1; route allocated: 1 of 10; total allocated: 1 of 10]
*** : Connection request: [route: {s}->https://api.github.com:443]
    [total kept alive: 1; route allocated: 1 of 10; total allocated: 1 of 10]
*** : Connection leased: [id: 0][route: {s}->https://api.github.com:443]
    [total kept alive: 0; route allocated: 1 of 10; total allocated: 1 of 10]
```

If your application is communicating with multiple hosts, you should adjust the `defaultMaxPerRoute` based on the number of the host you connect. However, there is a possibility that you will connect some HTTP services fewer than others, therefore it could make sense to define the `defaultMaxPerRoute` per host. `PoolingHttpClientConnectionManager` has `setMaxPerRoute(HttpRoute route, int max)` method, which accepts HTTP route for the request to be made and the maximum number of connection for that route particularly. You can refer to the following URL for more information about Apache HttpClient and connection management:

*https://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html[13]

To summarize, HTTP connection pool is a good way to gain performance. It lowers CPU and memory usage and reduces latency hence number of TCP connections that are established will be decreased.

[13] https://hc.apache.org/httpcomponents-client-ga/tutorial/html/connmgmt.html