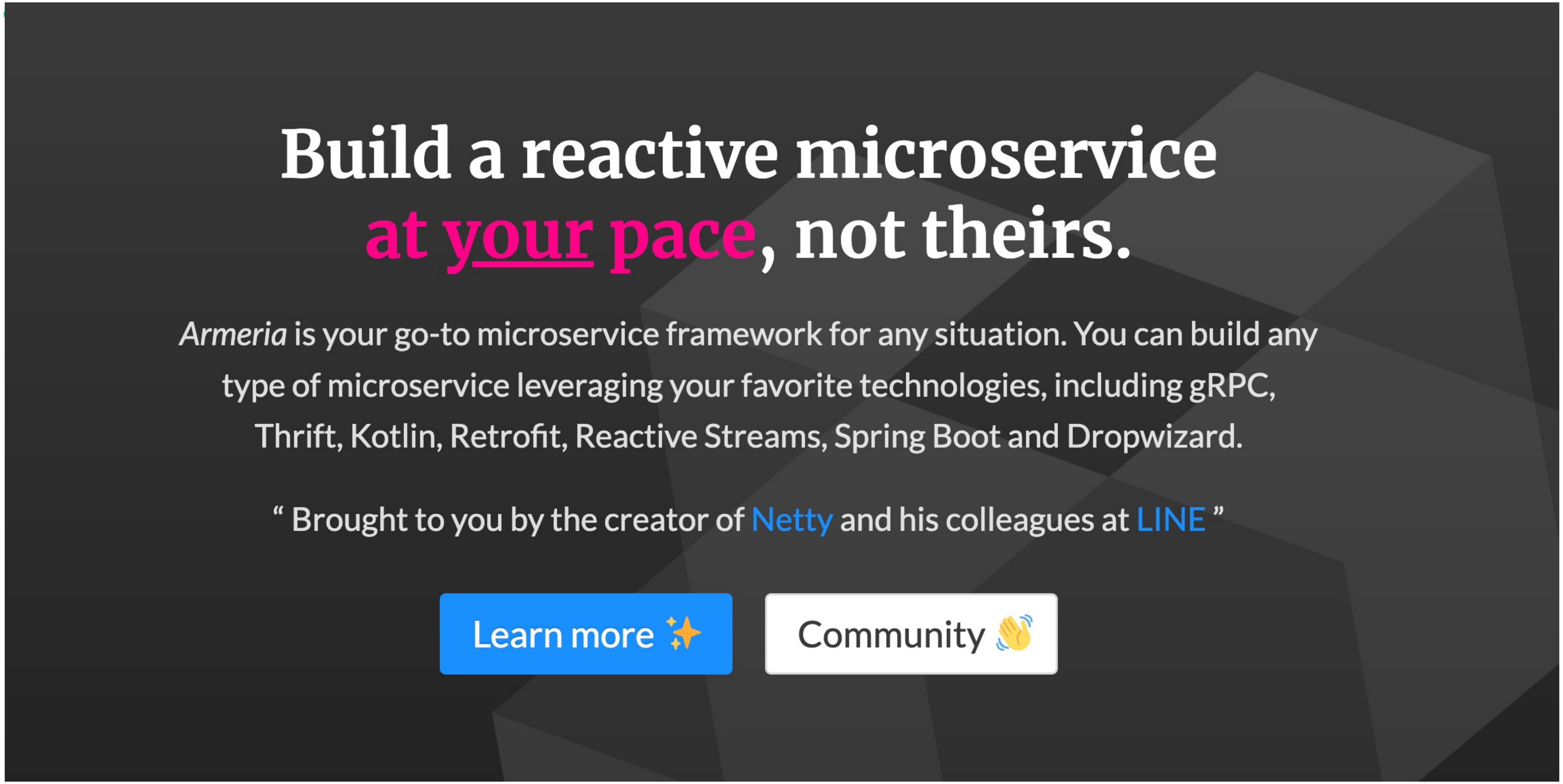
INFCON 2022

10만 connection 그까이꺼, Armeria 서버 한 대면 끝!

LINE+ 송민우, 엄익훈, 이한남

Before we get into...

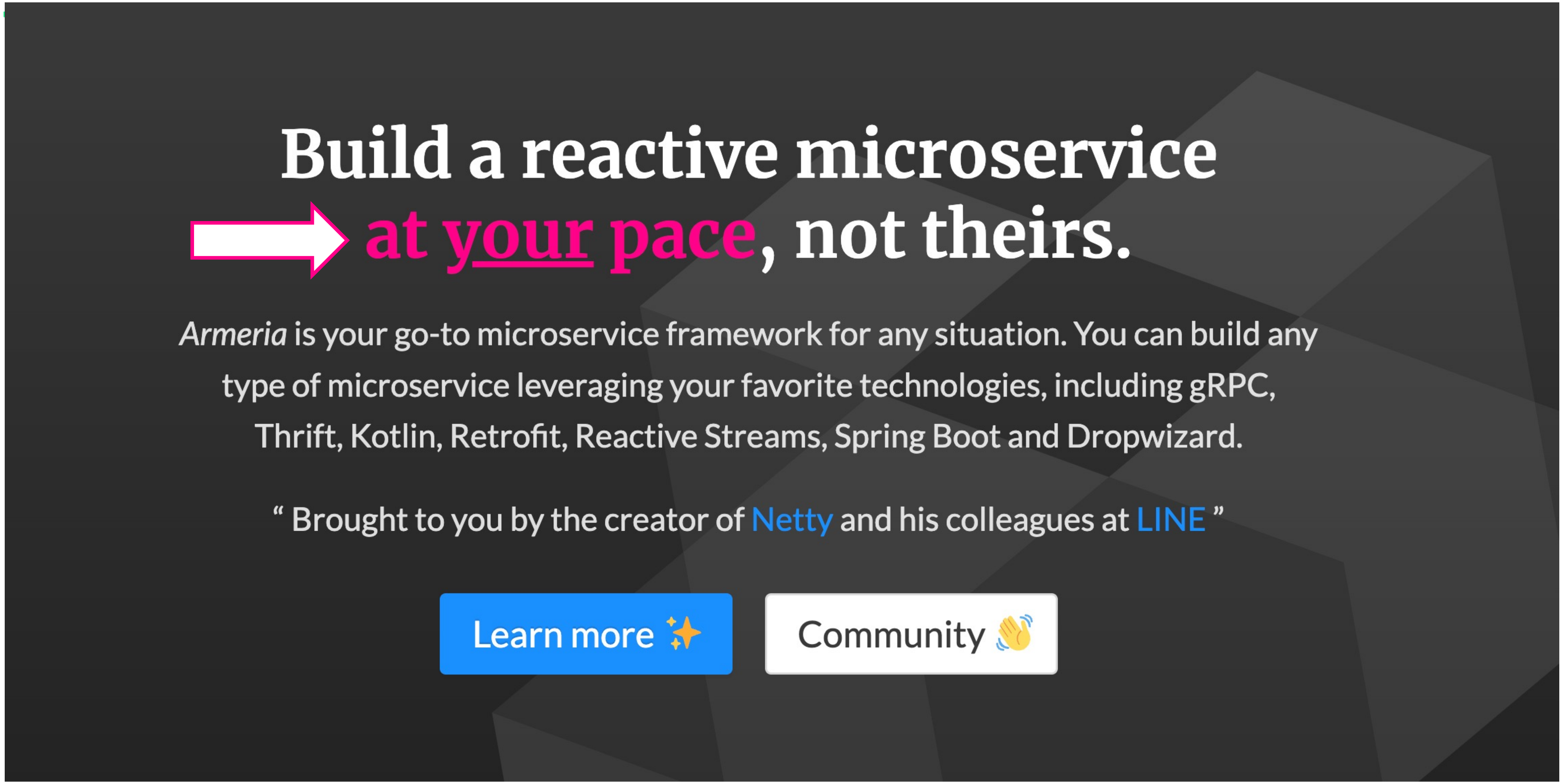
```
$ git clone https://github.com/minwoox/infcon-armeria.git

$ cd infcon-armeria

$ ./gradlew build
```



Build a reactive microservice at your pace, not theirs.

Armeria is your go-to microservice framework for any situation. You can build any type of microservice leveraging your favorite technologies, including gRPC, Thrift, Kotlin, Retrofit, Reactive Streams, Spring Boot and Dropwizard.

“ Brought to you by the creator of Netty and his colleagues at LINE ”



Build a reactive microservice
at your pace, not theirs.

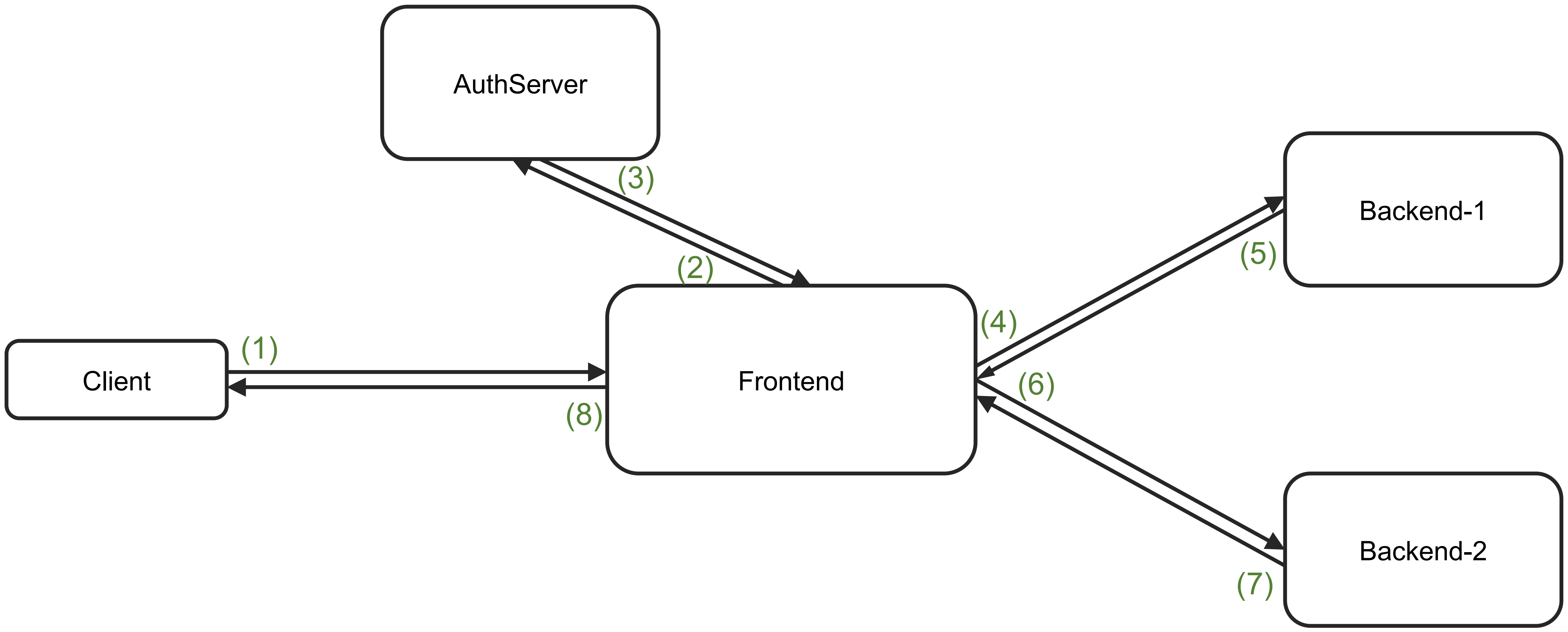
Armeria is your go-to microservice framework for any situation. You can build any type of microservice leveraging your favorite technologies, including gRPC, Thrift, Kotlin, Retrofit, Reactive Streams, Spring Boot and Dropwizard.

“ Brought to you by the creator of Netty and his colleagues at LINE ”

Things I wish we had known in advance

- 비동기 서버 그까이꺼, Request Scoping만 알면 끝!
- Armeria로 Reactive Streams와 놀자! - 1
- Armeria로 Reactive Streams와 놀자! – 2

Through this hands-on, we are going to build...



If you are new to an asynchronous server

- You are going to learn the basic principle of an asynchronism.

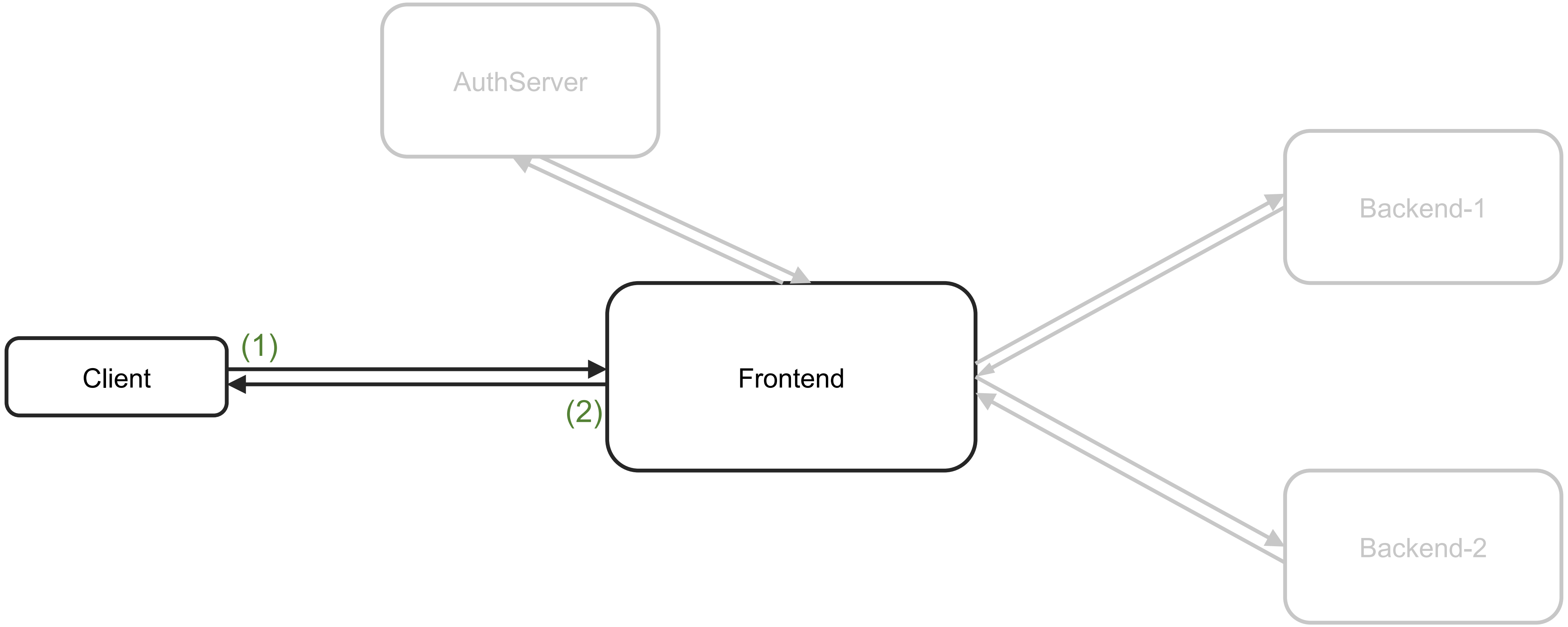
If you are new to Armeria

- You are going to learn some of Armeria APIs.

If you’ve already used Armeria

- Please, go out and listen to another session. 😆

Hello, Armeria!



Hello, Armeria!

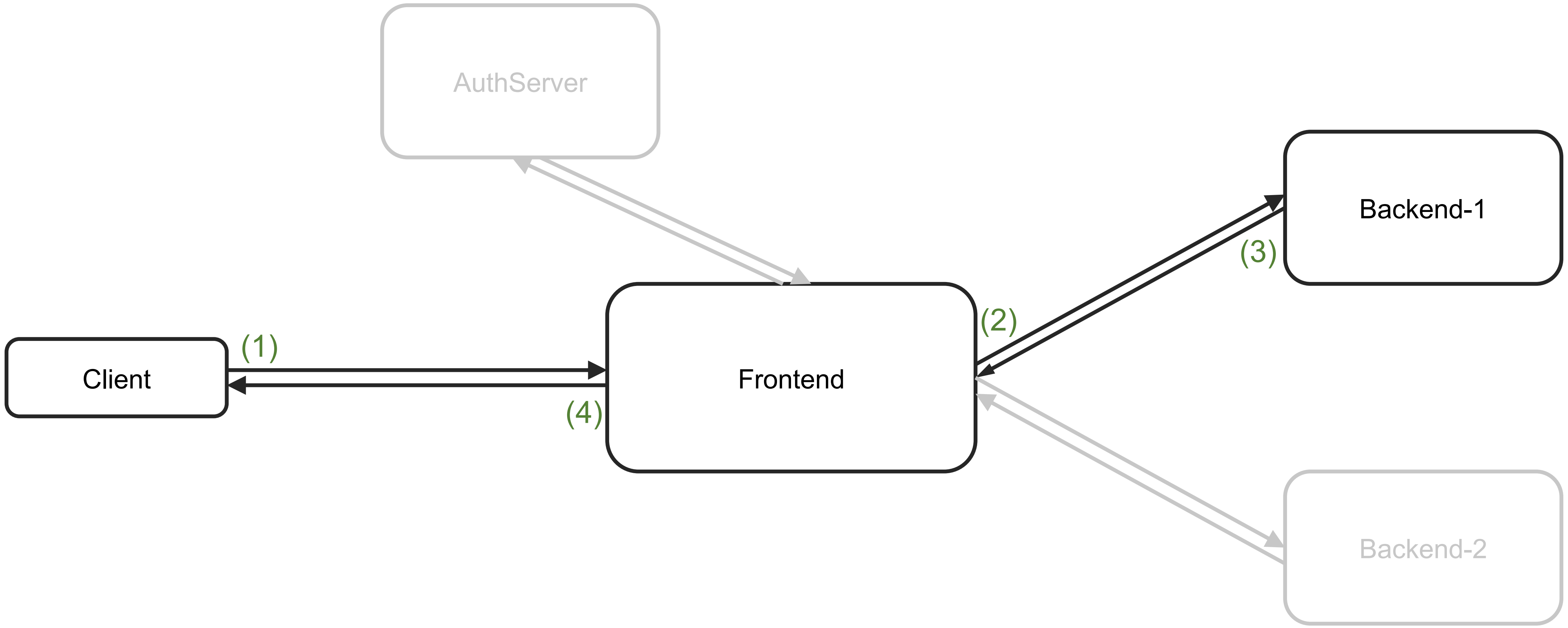
```java
public static void main(String[] args) {
    final ServerBuilder serverBuilder = Server.builder();
    final Server server =
            serverBuilder.http(8080)
                         .service("/infcon", (ctx, req) -> {
                             return HttpResponse.of("Hello, Armeria!");
                         })
                         .build();
    server.start().join();
}
```

```
$ curl http://127.0.0.1:8080/infcon
```

Hello, Armeria!

```
public final class HelloService implements HttpService {

    @Override
    public HttpResponse serve(
            ServiceRequestContext ctx, HttpRequest req) throws Exception {
        return HttpResponse.of("Hello, Armeria!");
    }
}
```

Backend that responds slowly



Backend that responds slowly

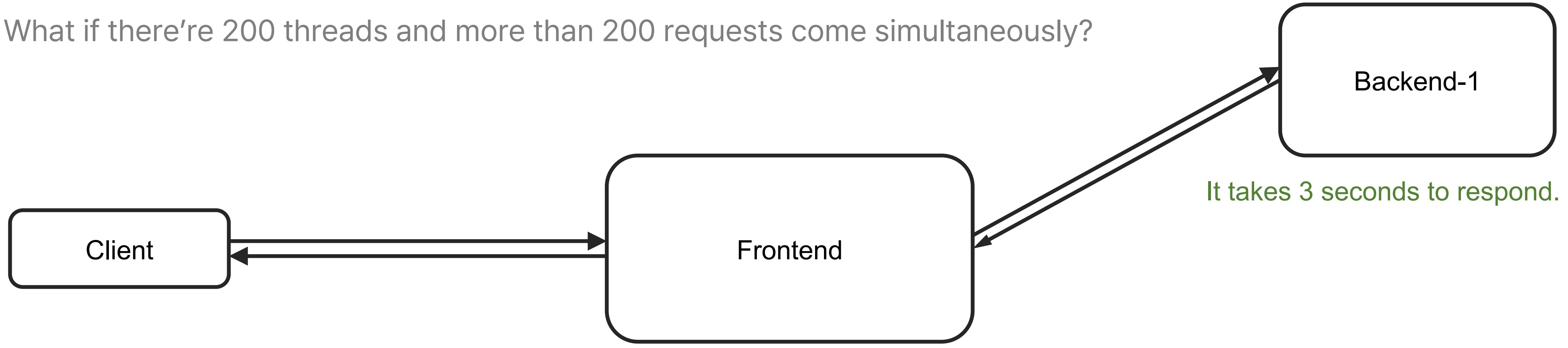
```
private Backend(String name, int port) {
    server = Server.builder()
                   .http(port)
                   .service("/foo", (ctx, req) -> {
                       return HttpResponse.delayed(HttpResponse.of("response from: " + name),
                                                   Duration.ofSeconds(3));
                   })
                   .build();

}
```

Let’s do the test!

```
@Test
void backend() {
    // response from: foo
    final Backend foo = Backend.of("foo", 9000);
    foo.start();

    final WebClient webClient = WebClient.of("http://127.0.0.1:9000");
    final HttpResponse httpResponse = webClient.get("/foo");
}
```

`HttpResponse` is just a wrapper

- Which means that it does not have anything when it's created.
- If it's not, the thread must wait 3 seconds to have the response
  which is synchronous.
- What if there're 200 threads and more than 200 requests come simultaneously?



Aggregating the response

- An HTTP response consists of frames that are not sent at once.
- Have to handle the frames one by one or just aggregate it.

```
HTTP/1.1 200 OK\r\n
Content-Length: 15\r\n
Content-Type: text/plain\r\n
…
Date: Thu, 25 Aug 2022 19:29:07 GMT\r\n
\r\n
Hello, Armeria!
```

Header frame

Data frame

Don’t block the event loop

- Let callbacks do the work.

Let’s do the test!

```java
@Test
void backend() {
    final Backend foo = Backend.of("foo", 9000);
    foo.start();

    final WebClient webClient = WebClient.of("http://127.0.0.1:9000");
    final HttpResponse httpResponse = webClient.get("/foo");
    final CompletableFuture<AggregatedHttpResponse> future = httpResponse.aggregate();
    final AggregatedHttpResponse aggregatedRessponse = future.join();
    System.err.println(aggregatedRessponse.contentUtf8());
}
```

Relaying the response

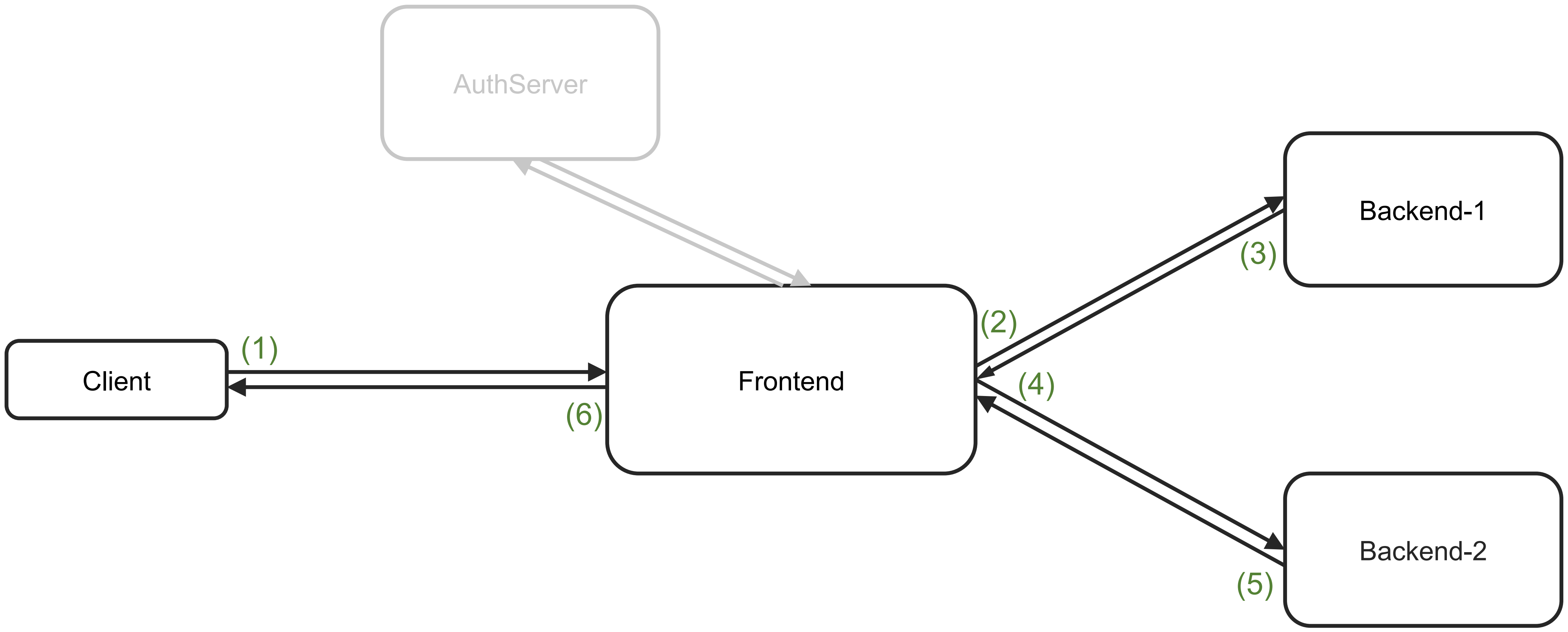
```
public static void main(String[] args) {
    final Backend foo = Backend.of("foo", 9000);
    foo.start();
    final WebClient fooClient = WebClient.of("http://127.0.0.1:9000");
    …
    serverBuilder.service("/infcon", new MyService(fooClient))
    …
}
```

Relaying the response

```java
public final class MyService implements HttpService {

    private final WebClient fooClient;

    public MyService(WebClient fooClient) {
        this.fooClient = fooClient;
    }

    @Override
    public HttpResponse serve(ServiceRequestContext ctx, HttpRequest req) throws Exception {
        return fooClient.get("/foo");
    }
}
```

```
$ curl http://127.0.0.1:8080/infcon
```

Composing responses



Before composing, how do we aggregate?

```java
@Override
public HttpResponse serve(ServiceRequestContext ctx, HttpRequest req) throws Exception {
    return HttpResponse.from(
            fooClient.get("/foo").aggregate().handle((fooResponse, cause1) -> {
        return HttpResponse.of(fooResponse.contentUtf8());
    }));
}
// Or
@Override
public HttpResponse serve(ServiceRequestContext ctx, HttpRequest req) throws Exception {
    final CompletableFuture<HttpResponse> future = new CompletableFuture<>();
    fooClient.get("/foo").aggregate().thenAccept(
            fooResponse -> future.complete(HttpResponse.of(HttpStatus.OK,
                                                           MediaType.PLAIN_TEXT_UTF_8,
                                                           fooResponse.contentUtf8())));
    return HttpResponse.from(future);
}
```

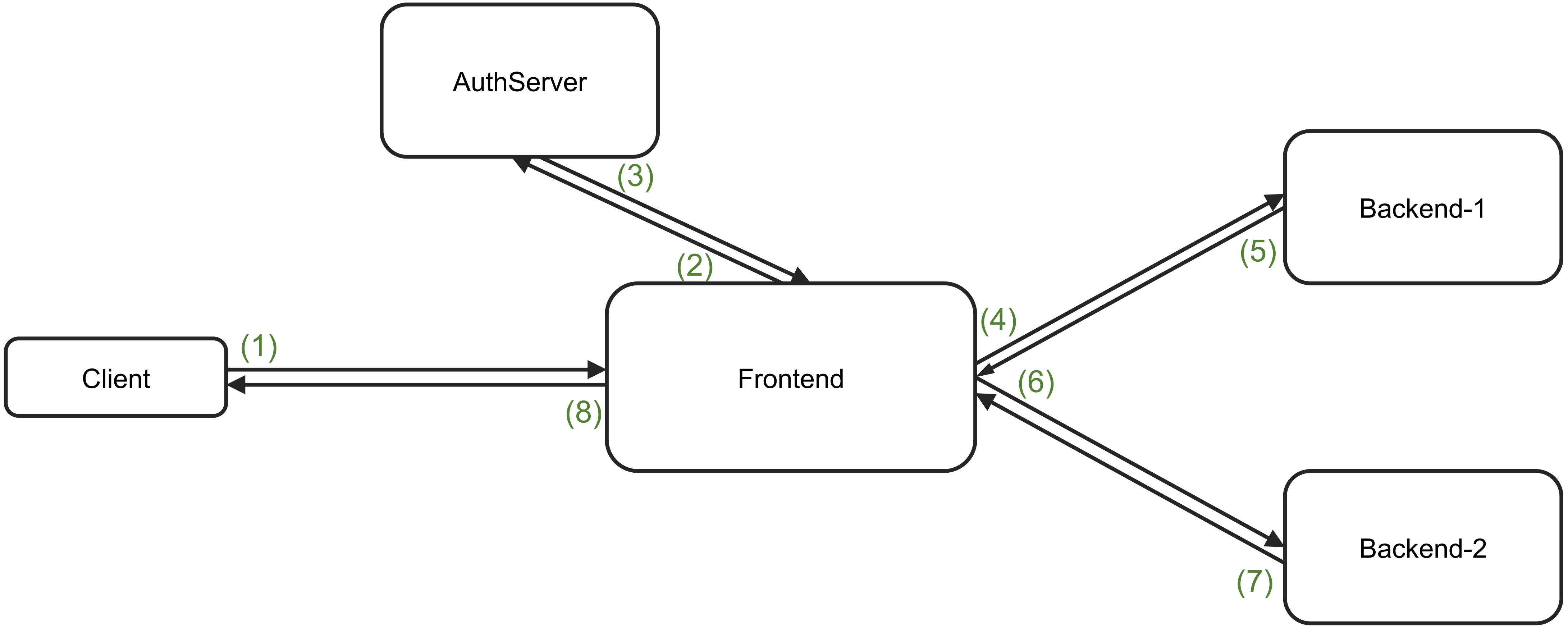
Composing responses

```java
public static void main(String[] args) {
    final Backend foo = Backend.of("foo", 9000);
    foo.start();
    final Backend bar = Backend.of("bar", 9001);
    bar.start();

    final WebClient fooClient = WebClient.of("http://127.0.0.1:9000");
    final WebClient barClient = WebClient.of("http://127.0.0.1:9001");

    final ServerBuilder serverBuilder = Server.builder();
    final Server server =
            serverBuilder.http(8080)
                         .service("/infcon", new MyService(fooClient, barClient))
                         .build();
    server.start().join();
}
```

Composing responses

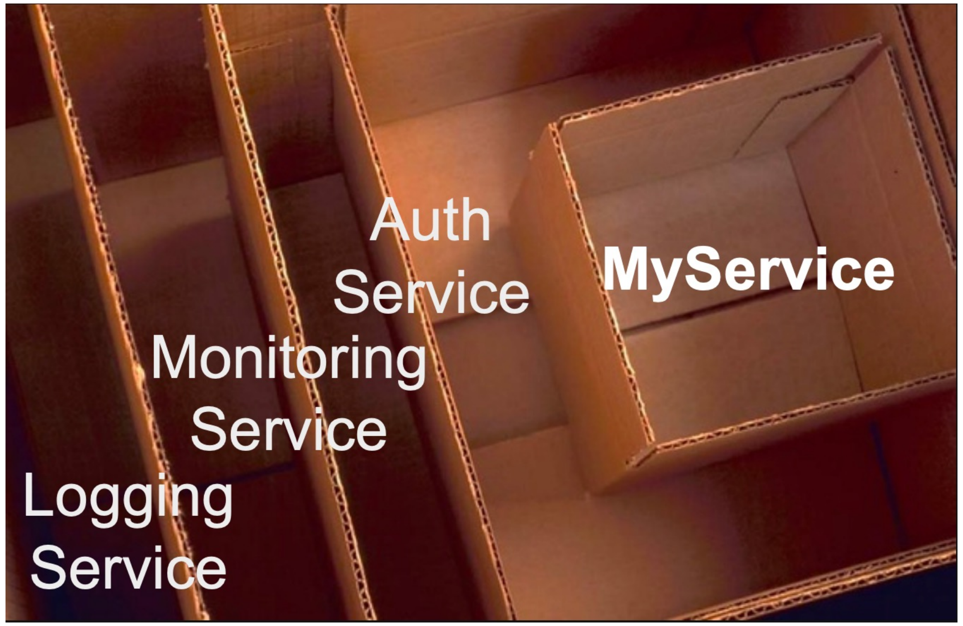
```java
public final class MyService implements HttpService {

    private final WebClient fooClient;
    private final WebClient barClient;

    public MyService(WebClient fooClient, WebClient barClient) {
        this.fooClient = fooClient;
        this.barClient = barClient;
    }

    @Override
    public HttpResponse serve(ServiceRequestContext ctx, HttpRequest req) throws Exception {
        ...
    }
}
```

Decorator



Decorator

- A decorating service (or decorator) is a service that wraps another service to intercept an incoming request or an outgoing response.
- An implementation of the decorator pattern
- Core features such as logging, metrics and distributed tracing are implemented as decorators.



https://www.slideshare.net/JulieKim1/armeriaworkshop2019-openchat-julie

AuthDecorator

```
@Override
public HttpResponse serve(
        HttpService delegate, ServiceRequestContext ctx, HttpRequest req) throws Exception {
    final CompletableFuture<HttpResponse> future = new CompletableFuture<>();
    authClient.get("/auth").aggregate().whenComplete((aggregatedHttpResponse, cause) -> {
        try {
            future.complete(delegate.serve(ctx, req));
        } catch (Throwable t) {
            future.completeExceptionally(t);
        }
    });
    return HttpResponse.from(future);
}
```

github.com/line/armeria

armeria.dev

@armeria_project line/armeria