DISTRIBUTED SYSTEMS

Lab 2

Nuno Preguiça, João Leitão, Pedro Fouto, Luís Silva

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- Know how to develop a REST Client in Java (using JAX-RX)
- Use Docker to test your service using your clients

Goals

In the end of this lab you should be able to:

- **Understand what a WebService REST is**
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- Know how to develop a REST Client in Java (using JAX-RX)
- Use Docker to test your service using your clients

REST : REPRESENTATIONAL STATE TRANSFER

Architectural Pattern to access Information

Fundamental Approach:

An application is perceived as a collection of resources.

The key implications of this are:

- A resource is identified by a URI/URL
- The URL returns a document with a representation of the resource
- A URL can refer to a collection of resources
- It is possible to refer to other resources (from a resource) using links

REST : REPRESENTATIONAL STATE TRANSFER EXAMPLE

Consider an application that is used to manage contact cards.

- A contact card is a resource and each contact card has an URL associated.
- The URL of a card will return a representation of that card (could be a textual representation of the fields of the card) – name of the person, phone, e-mail, postal address – but it could also be a binary representation.
- An URL can represent the whole collection of contact cards managed by an application.
- A contact card could contain the URL of another card, for instance to refer to the spouse of that person.

REST PROTOCOL

A client-server protocol that is **stateless**: each request contains all the information that is necessary to process the request.

- This implies that the server does not need to keep track of relations among different requests
- It makes the interaction pattern of systems using rest simple
- It allows to do transparent caching

REST PROTOCOL

The REST interface is **uniform**: all resources are accessed by a set of well-defined HTTP operations:

- **POST**: Creates a new resource
- **GET**: Obtains (a representation of) an existing resource
- **PUT**: Updates or Replaces an existing resource
- **DELETE**: Eliminates an existing resource

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- **Know how to develop a WS REST and Server in Java (using JAX-RS)**
- Know how to develop a REST Client in Java (using JAX-RX)
- Use Docker to test your service using your clients

Development of a web service REST in Java

In the Distributed Systems Course we are using the Jersey (JAX-RS) framework, which highly simplifies the development of REST services in Java.

- When using this framework, we instrument our code through simple annotations in our Java code (e.g., @PATH, @GET, @POST, @DELETE, …)
- Java Reflection is taken advantage by the Jersey runtime to derive code automatically based on those annotations.

Want to know more? https://eclipse-ee4j.github.io/jersey/

DEVELOPMENT OF A WEB SERVICE REST IN JAVA

There are a few dependencies that our code will have. As discussed last week these will be handled by Maven. The dependencies are inserted in the pom.xml file:

```xml
<dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.30.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.30.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-server</artifactId>
    <version>2.30.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-jaxb</artifactId>
    <version>2.30.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-jdk-http</artifactId>
    <version>2.30.1</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>2.30.1</version>
  </dependency>
</dependencies>
```

Development of a web service REST in Java

We are going to show how to develop a Web Service REST by example.

Our example, *not accidentally*, is based on the construction of a simple mail service (and example clients).

- In this service, mails are messages sent by one user, to one or more users. Messages have multiple fields such as a subject and content.
- Messages are stored in our server (and identified by a positive long integer). They are also associated with the inbox of their destinations (recipients of the message).
- Messages are going to be our main resource in this example (we do not have other types of explicit resources such as users for simplicity).

Modelling our Message Resource as a Java Class

Standard Java Class

```
package sd1920.aula2.api;
import …
public class Message {
        private String sender;
        private Set<String> destination;
        private long creationTime;
        private String subject;
        private byte[] contents;

        public Message() {
                this.sender = null;
                this.destination = new HashSet<String>();
                this.creationTime = System.currentTimeMillis();
                this.subject = null;
                this.contents = null;
        }

        public Message(String sender, String destination, String subject, byte[] contents) {
                …
        }
        public String getSender() { … }

        public void setSender(String sender) { … }
```

Modelling our Message Resource as a Java Class

Standard Java Class

```
package sd1920.aula2.api;
import …
public class Message {
        private String sender;
        private Set<String> destination;
        private long creationTime;
        private String subject;
        private byte[] contents;

        public Message() {
                this.sender = null;
                this.destination = new HashSet<String>();
                this.creationTime = System.currentTimeMillis();
                this.subject = null;
                this.contents = null;
        }

        public Message(String sender, String destination, String subject, byte[] contents) {
                …
        }
        public String getSender() { … }

        public void setSender(String sender) { … }
…
```

Private Fields (but you have to create standard getters and setters)

Modelling our Message Resource as a Java Class

Standard Java Class

```
package sd1920.aula2.api;
import …
public class Message {
		private String sender;
		private Set<String> destination;
		private long creationTime;
		private String subject;
		private byte[] contents;

		public Message() {
				this.sender = null;
				this.destination = new HashSet<String>();
				this.creationTime = System.currentTimeMillis();
				this.subject = null;
				this.contents = null;
		}
		public Message(String sender, String destination, String subject, byte[] contents) {
				…
		}
		public String getSender() { … }

		public void setSender(String sender) { … }
…
```

You can have any number of constructors…

MODELLING OUR MESSAGE RESOURCE AS A JAVA CLASS

Standard Java Class

```
package sd1920.aula2.api;
import …
public class Message {
        private String sender;
        private Set<String> destination;
        private long creationTime;
        private String subject;
        private byte[] contents;

        public Message() {
                this.sender = null;
                this.destination = new HashSet<String>();
                this.creationTime = System.currentTimeMillis();
                this.subject = null;
                this.contents = null;
        }

        public Message(String sender, String destination, String subject, byte[] contents) {
                …
        }
        public String getSender() { … }

        public void setSender(String sender) { … }
…
```

But you should have a default constructor without arguments.

MODELLING OUR MESSAGE RESOURCE AS A JAVA CLASS

```
package sd1920.aula2.api;
import …
public class Message {
            private String sender;
            private Set<String> destination;
            private long creationTime;
            private String subject;
            private byte[] contents;

            public Message() {
                        this.sender = null;
                        this.destination = new HashSet<String>();
                        this.creationTime = System.currentTimeMillis();
                        this.subject = null;
                        this.contents = null;
            }

            public Message(String sender, String destination, String subject, byte[] contents) {
                        …
            }
            public String getSender() { … }

            public void setSender(String sender) { … }
…
```

Standard Java Class

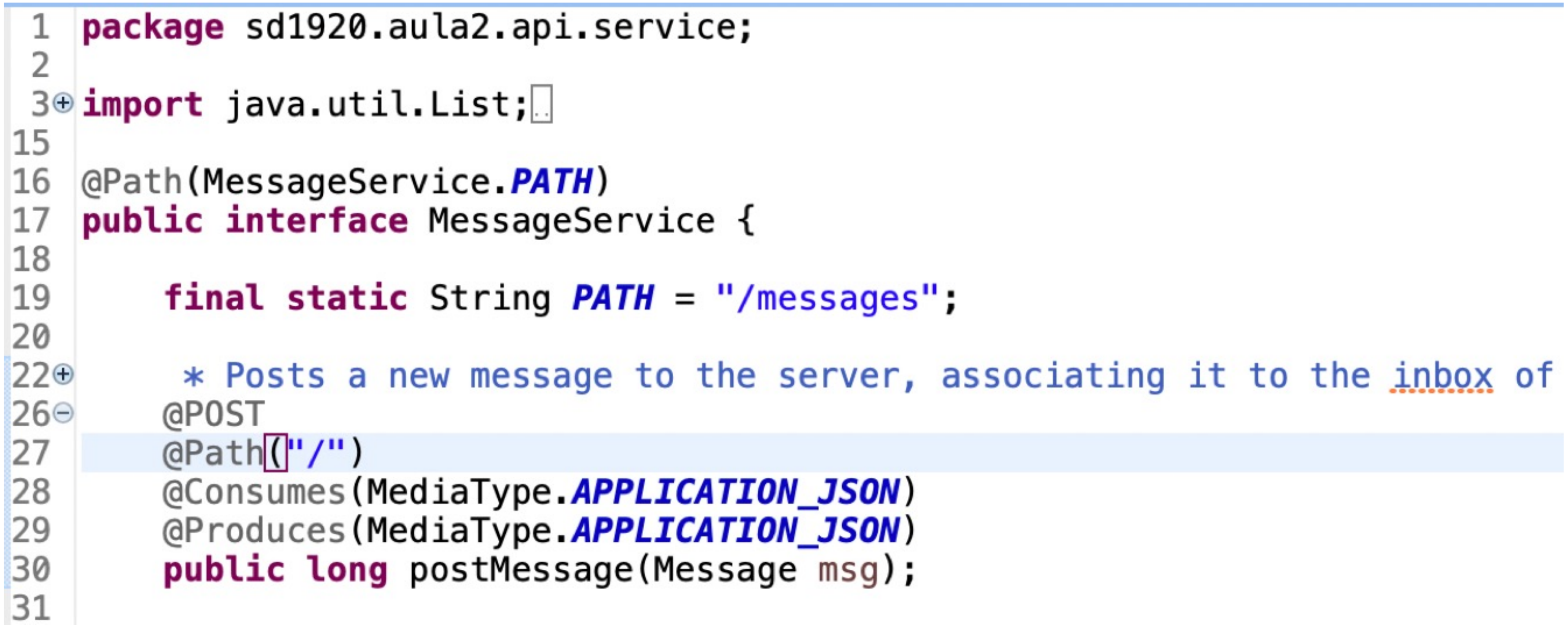
But you should have a default constructor without arguments.

Default constructor and getters/setters are important to allow the serialization and deserialization of this class over the network.

DEFINING THE SERVICE INTERFACE

```
package sd1920.aula2.api.service;
import …
@Path(MessageService.PATH)
public interface MessageService {
    final static String PATH = "/messages";

    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

    @GET
    @Path("/{mid}")
    @Produces(MediaType.APPLICATION_JSON)
    public Message getMessage(@PathParam("mid") long mid);

    @GET
    @Path("/{mid}/body")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public byte[] getMessageBody(@PathParam("mid") long mid);

    @GET
    @Path("/")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Message> getMessage(@QueryParam("user") String user);

    // MUST COMPLETE
    void deleteMessage(long mid);

    // MUST COMPLETE
    void removeFromUserInbox(String user, long mid);

}
```

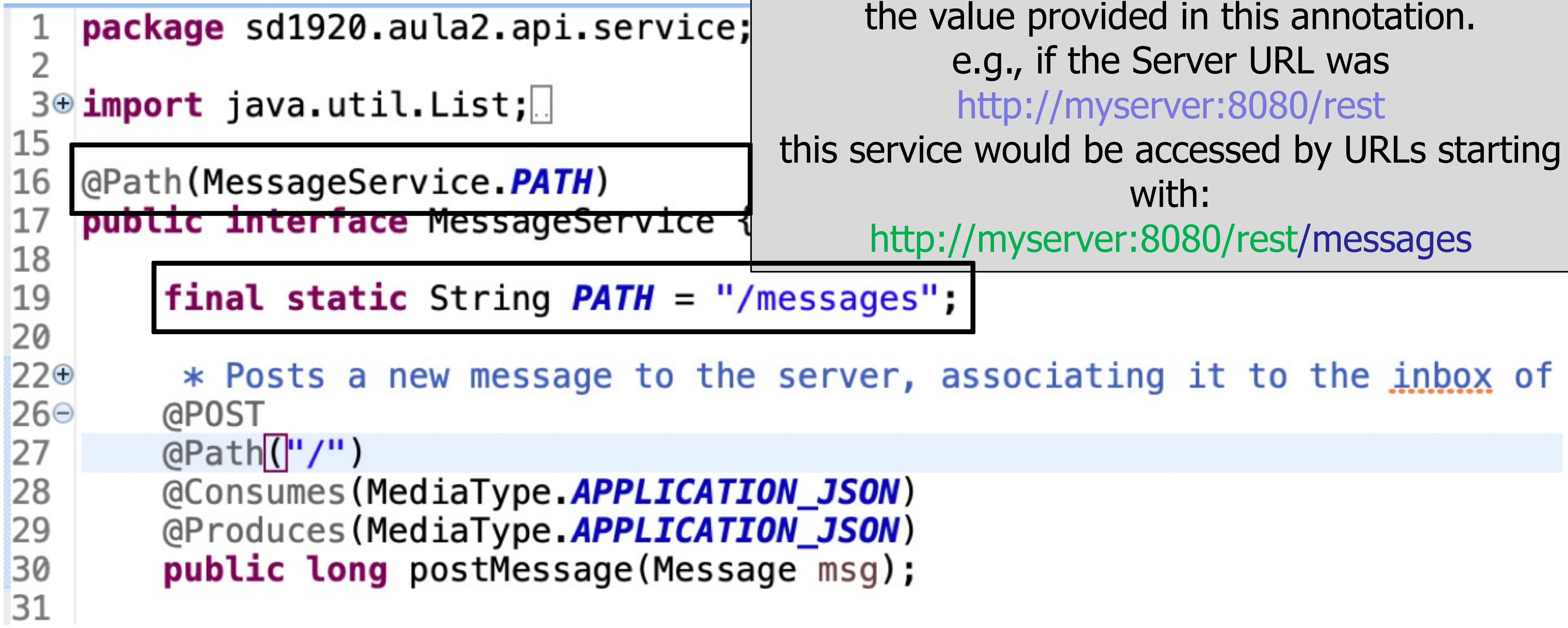
Defining the Service Interface



```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService {

    final static String PATH = "/messages";

     * Posts a new message to the server, associating it to the inbox of
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

```

Standard Java Interface enriched with Jersey annotations and identifying the methods supported by your service.

DEFINING THE SERVICE INTERFACE



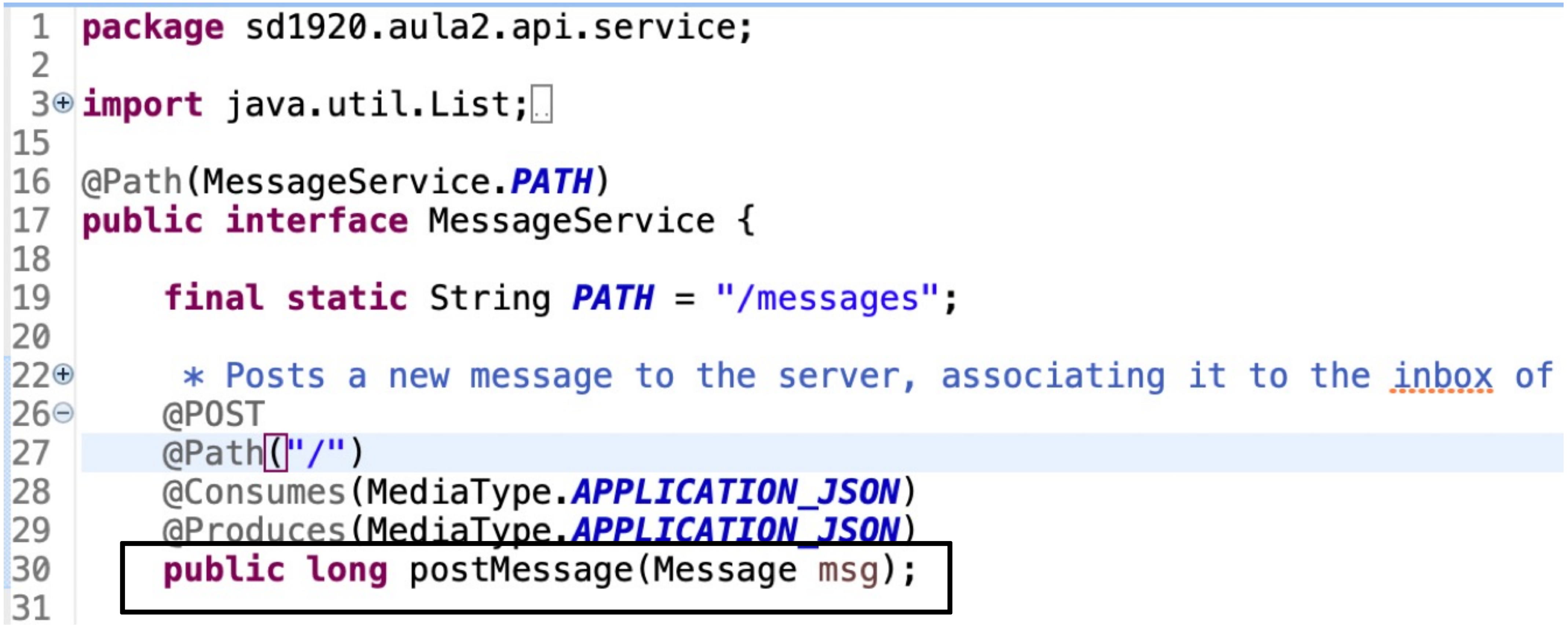
```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService {

    final static String PATH = "/messages";

     * Posts a new message to the server, associating it to the inbox of
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

```

@Path(STRING VALUE)
This will be used to define the URL used to access this service. It will be the Server URL + the value provided in this annotation.
e.g., if the Server URL was
http://myserver:8080/rest
this service would be accessed by URLs starting with:
http://myserver:8080/rest/messages

Standard Java Interface enriched with Jersey annotations and identifying the methods supported by your service.

DEFINING THE SERVICE INTERFACE

This method will allow to create (i.e., store) a new message in the server.

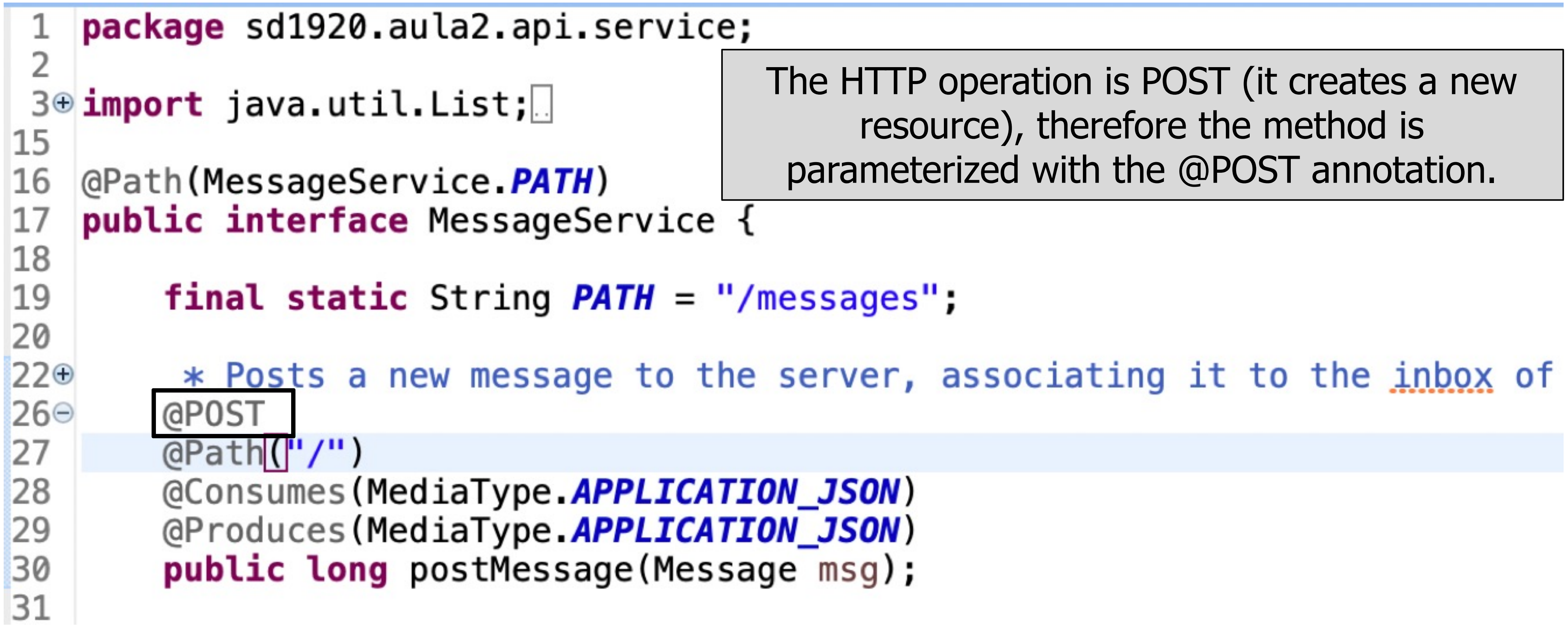


```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService {

    final static String PATH = "/messages";

     * Posts a new message to the server, associating it to the inbox of
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

```

Standard Java Interface enriched with Jersey annotations and identifying the methods supported by your service.

Defining the Service Interface

This method will allow to create (i.e., store) a new message in the server.

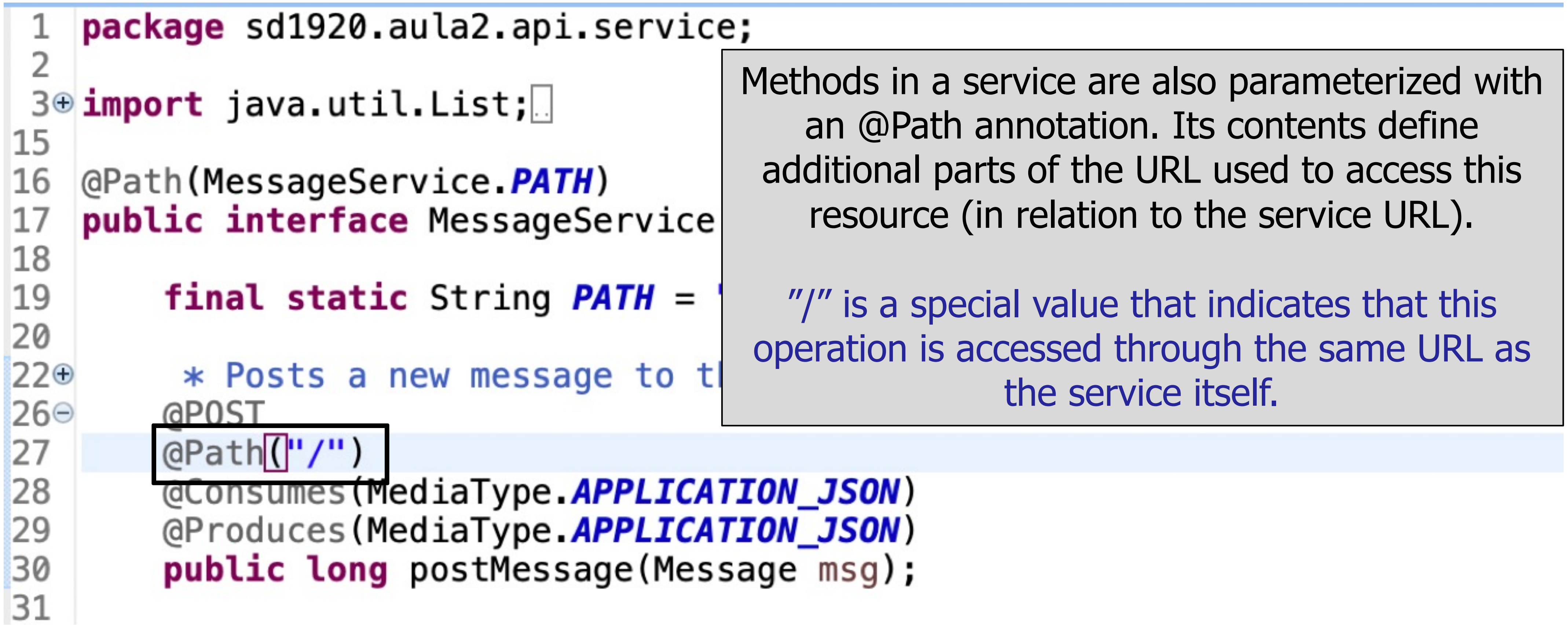


```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService {

    final static String PATH = "/messages";

     * Posts a new message to the server, associating it to the inbox of
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

```

The HTTP operation is POST (it creates a new resource), therefore the method is parameterized with the @POST annotation.

DEFINING THE SERVICE INTERFACE

This method will allow to create (i.e., store) a new message in the server.



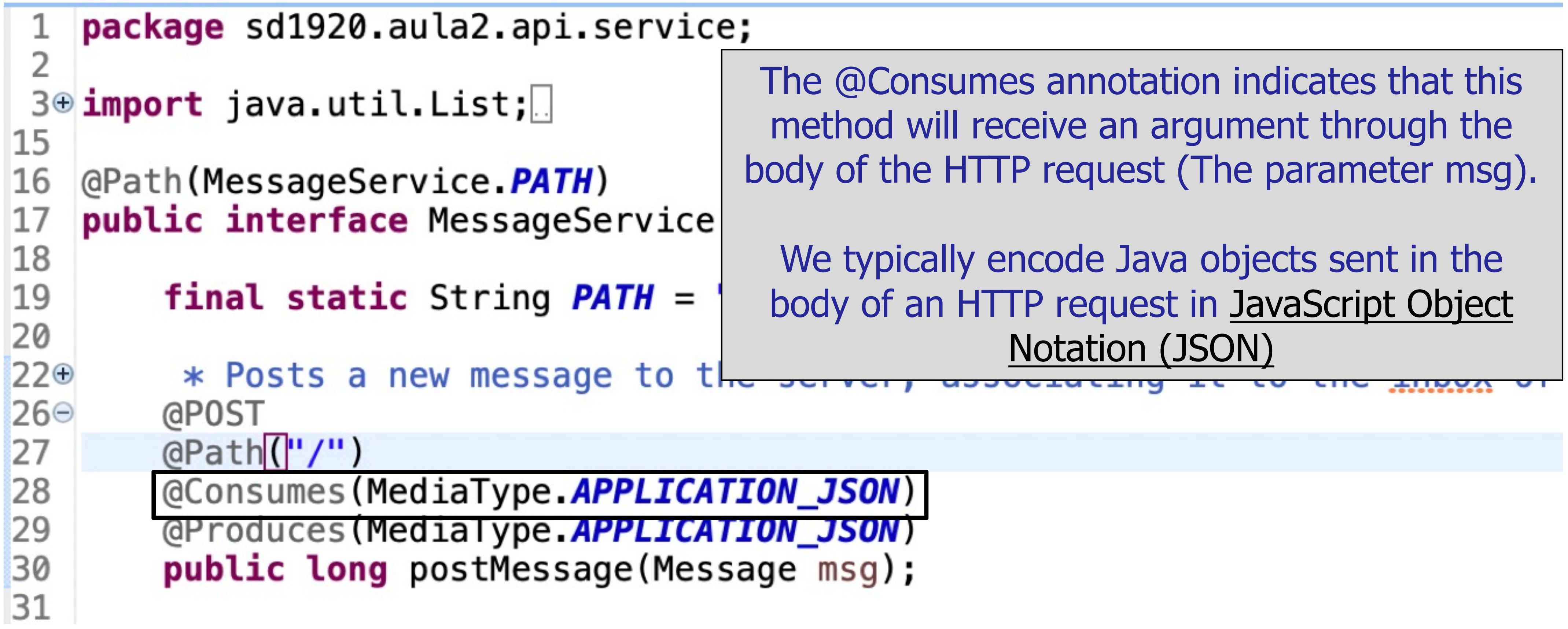
```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService

    final static String PATH = '

     * Posts a new message to tl
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

```

Methods in a service are also parameterized with an @Path annotation. Its contents define additional parts of the URL used to access this resource (in relation to the service URL).

"/" is a special value that indicates that this operation is accessed through the same URL as the service itself.

DEFINING THE SERVICE INTERFACE

This method will allow to create (i.e., store) a new message in the server.



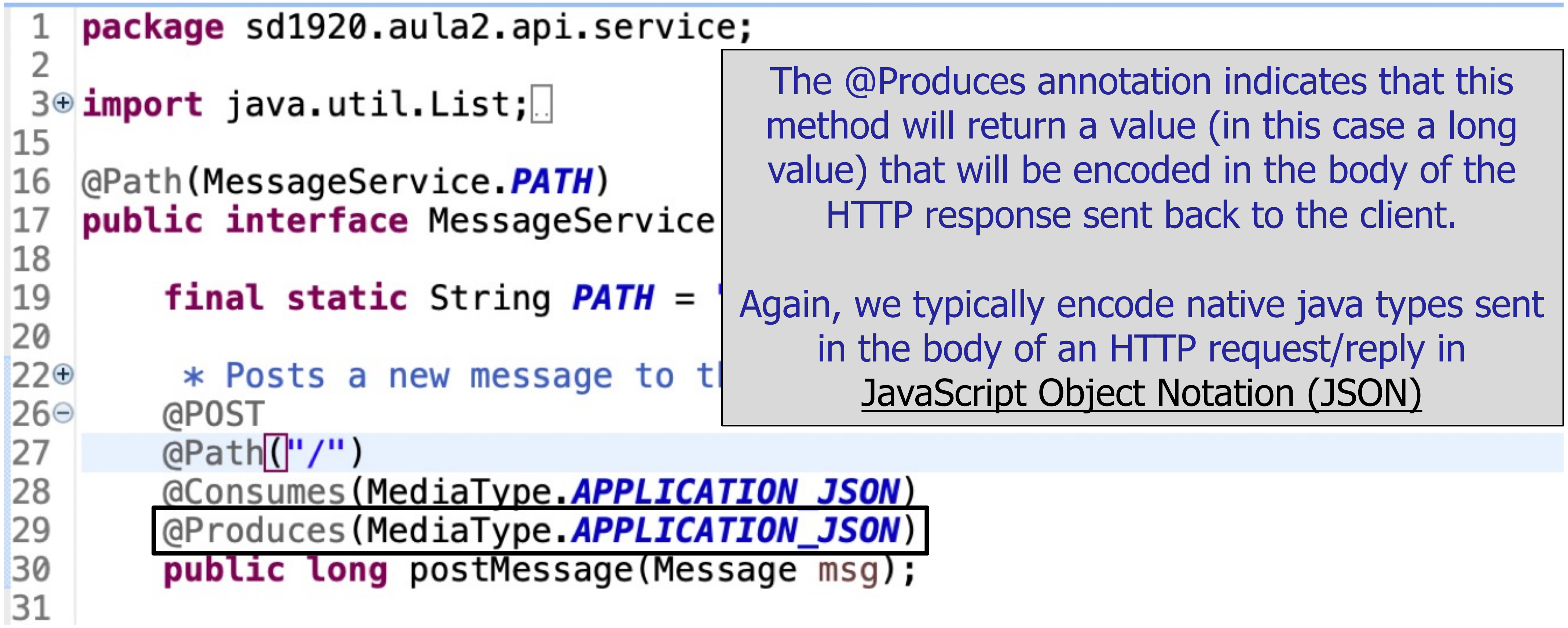
```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService

    final static String PATH = '

     * Posts a new message to the server, associating it to the inbox of
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);
```

The @Consumes annotation indicates that this method will receive an argument through the body of the HTTP request (The parameter msg).

We typically encode Java objects sent in the body of an HTTP request in JavaScript Object Notation (JSON)

DEFINING THE SERVICE INTERFACE

This method will allow to create (i.e., store) a new message in the server.



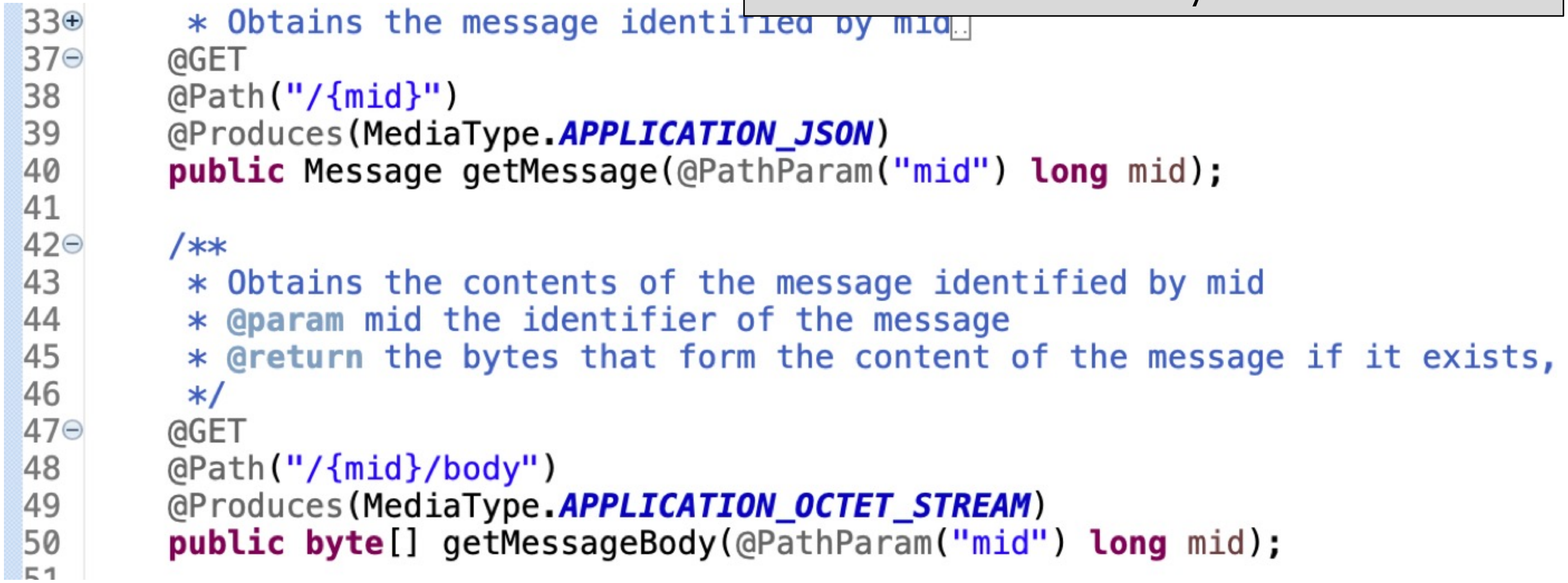
```
package sd1920.aula2.api.service;

import java.util.List;

@Path(MessageService.PATH)
public interface MessageService

    final static String PATH = '

     * Posts a new message to tl
    @POST
    @Path("/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public long postMessage(Message msg);

```

The @Produces annotation indicates that this method will return a value (in this case a long value) that will be encoded in the body of the HTTP response sent back to the client.

Again, we typically encode native java types sent in the body of an HTTP request/reply in JavaScript Object Notation (JSON)

DEFINING THE SERVICE INTERFACE

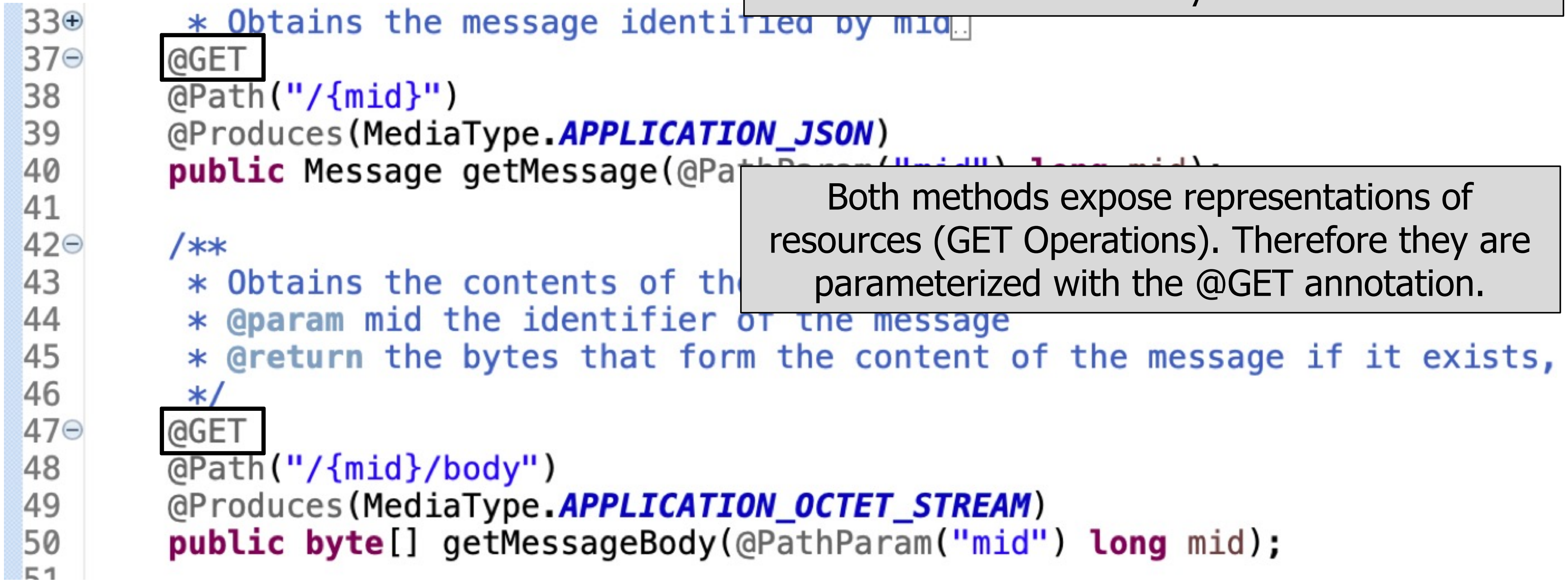
These methods will allow to, respectively, access an existing message in the server, and access the contents of the message (encoded as byte[]) directly.



```
     * Obtains the message identified by mid
    @GET
    @Path("/{mid}")
    @Produces(MediaType.APPLICATION_JSON)
    public Message getMessage(@PathParam("mid") long mid);

    /**
     * Obtains the contents of the message identified by mid
     * @param mid the identifier of the message
     * @return the bytes that form the content of the message if it exists,
     */
    @GET
    @Path("/{mid}/body")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public byte[] getMessageBody(@PathParam("mid") long mid);
```

Defining the Service Interface

These methods will allow to, respectively, access an existing message in the server, and access the contents of the message (encoded as byte[]) directly.

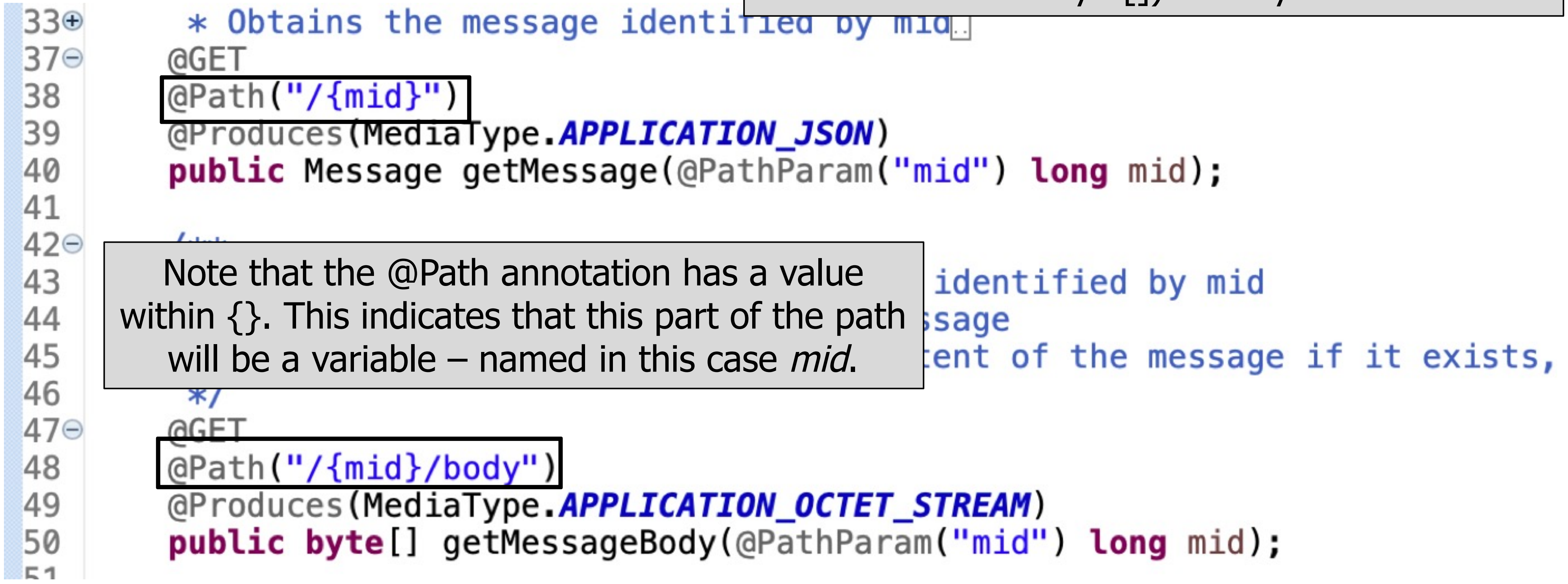


```
     * Obtains the message identified by mid
    @GET
    @Path("/{mid}")
    @Produces(MediaType.APPLICATION_JSON)
    public Message getMessage(@PathParam("mid") long mid);

    /**
     * Obtains the contents of th
     * @param mid the identifier of the message
     * @return the bytes that form the content of the message if it exists,
     */
    @GET
    @Path("/{mid}/body")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public byte[] getMessageBody(@PathParam("mid") long mid);
```

Both methods expose representations of resources (GET Operations). Therefore they are parameterized with the @GET annotation.

DEFINING THE SERVICE INTERFACE

These methods will allow to, respectively, access an existing message in the server, and access the contents of the message (which are encoded as byte[]) directly.

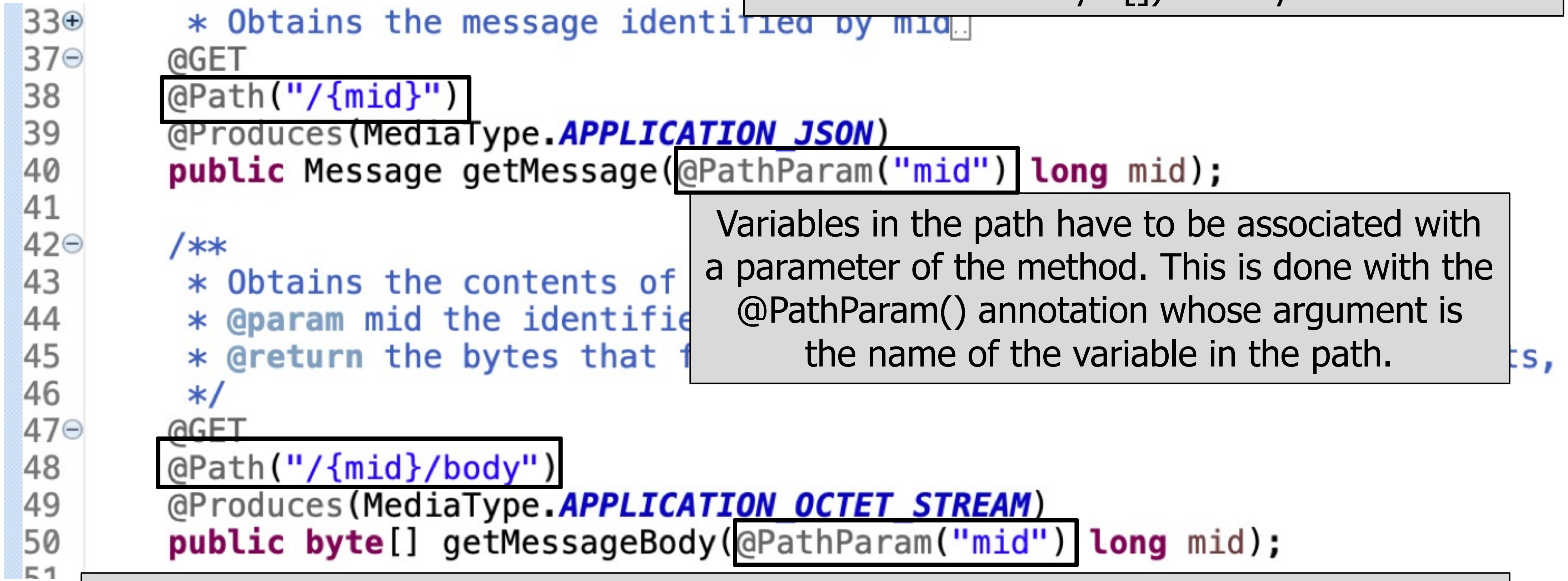


```
     * Obtains the message identified by mid
    @GET
    @Path("/{mid}")
    @Produces(MediaType.APPLICATION_JSON)
    public Message getMessage(@PathParam("mid") long mid);

    /**
                                    identified by mid
                                   ssage
                                   ent of the message if it exists,
     */
    @GET
    @Path("/{mid}/body")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public byte[] getMessageBody(@PathParam("mid") long mid);
```

Note that the @Path annotation has a value within {}. This indicates that this part of the path will be a variable – named in this case *mid*.

DEFINING THE SERVICE INTERFACE

These methods will allow to, respectively, access an existing message in the server, and access the contents of the message (which are encoded as byte[]) directly.



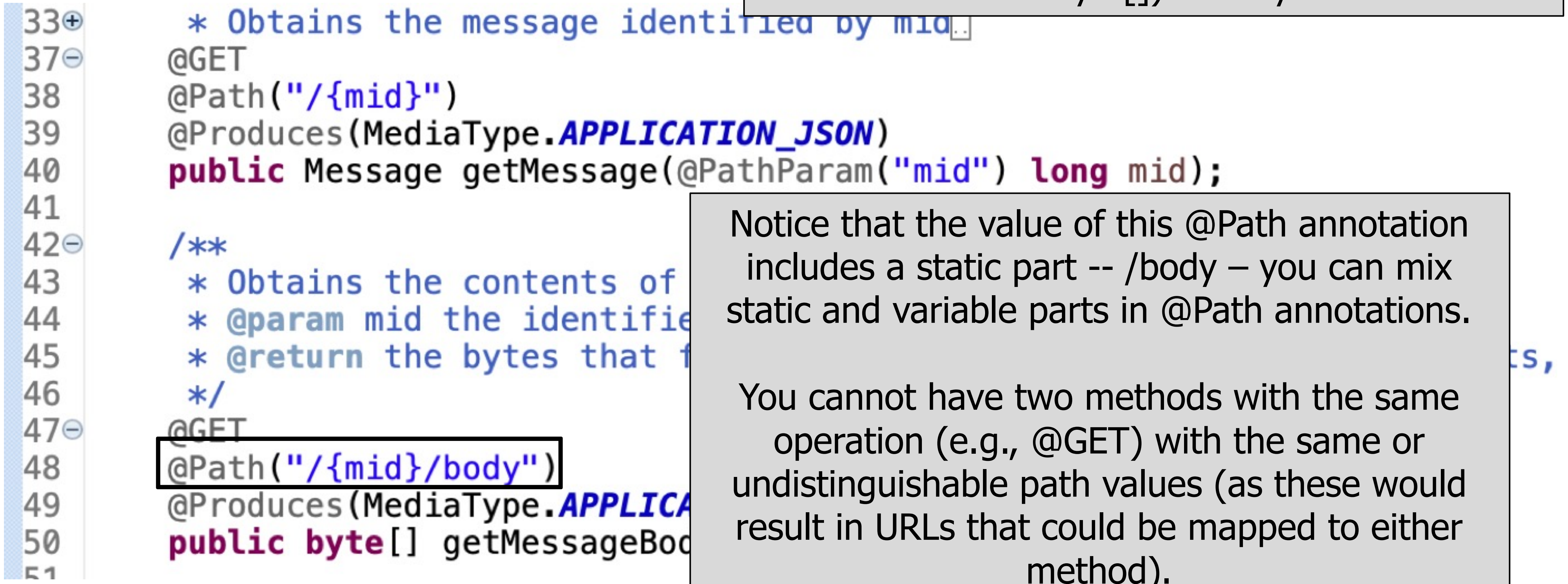
```
     * Obtains the message identified by mid
    @GET
    @Path("/{mid}")
    @Produces(MediaType.APPLICATION_JSON)
    public Message getMessage(@PathParam("mid") long mid);

    /**
     * Obtains the contents of
     * @param mid the identifie
     * @return the bytes that f                                  ts,
     */
    @GET
    @Path("/{mid}/body")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public byte[] getMessageBody(@PathParam("mid") long mid);
```

Variables in the path have to be associated with a parameter of the method. This is done with the @PathParam() annotation whose argument is the name of the variable in the path.

Jersey will process the URL automatically and assign it to that method parameter.
You can have multiple path variables mapped with @PathParam in the same path.
Only native types (including String) can be passed as Path parameters (i.e., Java classes and byte[] have to be passed through the body of the HTTP request using @Consumes)

DEFINING THE SERVICE INTERFACE

These methods will allow to, respectively, access an existing message in the server, and access the contents of the message (which are encoded as byte[]) directly.



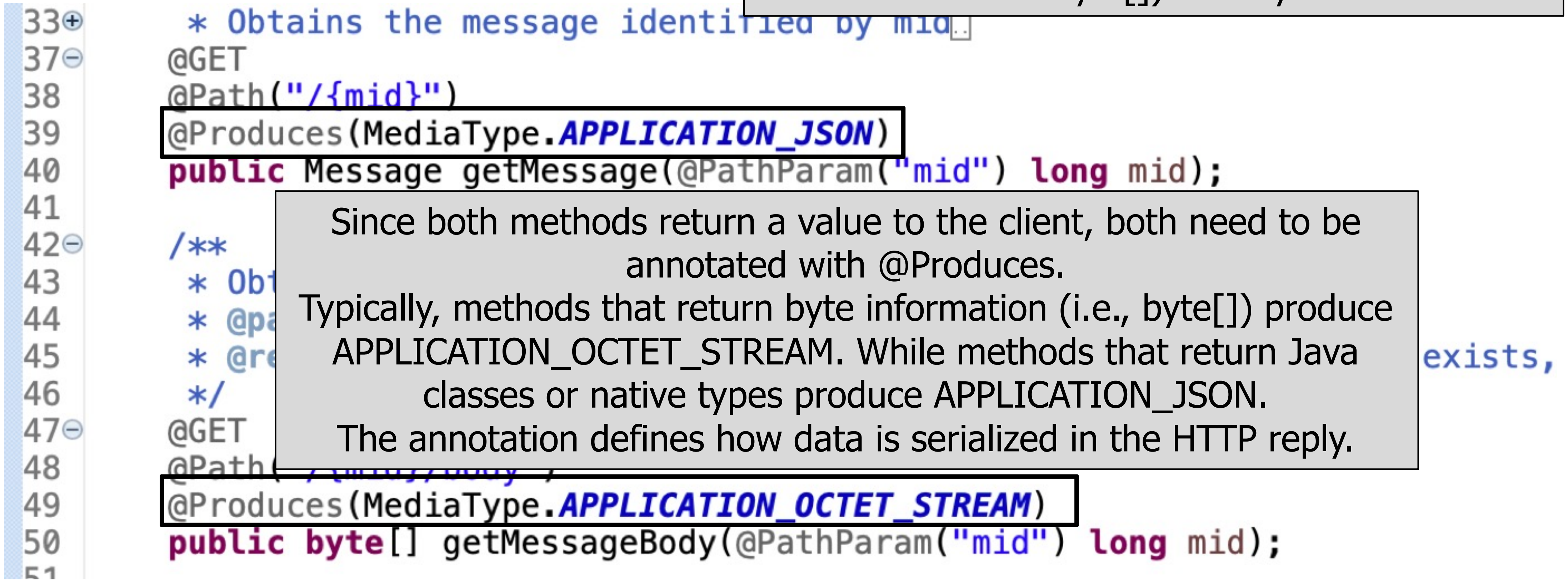
```
 * Obtains the message identified by mid
@GET
@Path("/{mid}")
@Produces(MediaType.APPLICATION_JSON)
public Message getMessage(@PathParam("mid") long mid);

/**
 * Obtains the contents of
 * @param mid the identifie
 * @return the bytes that f                               ts,
 */
@GET
@Path("/{mid}/body")
@Produces(MediaType.APPLICA
public byte[] getMessageBod
```

Notice that the value of this @Path annotation includes a static part -- /body – you can mix static and variable parts in @Path annotations.

You cannot have two methods with the same operation (e.g., @GET) with the same or undistinguishable path values (as these would result in URLs that could be mapped to either method).

DEFINING THE SERVICE INTERFACE

These methods will allow to, respectively, access an existing message in the server, and access the contents of the message (which are encoded as byte[]) directly.

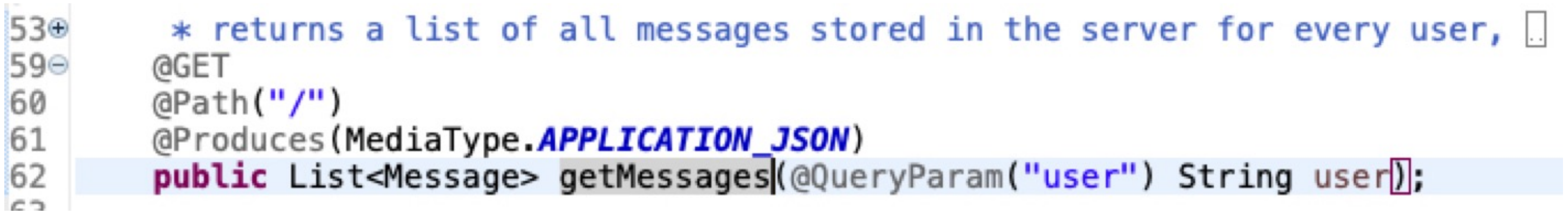


```
 * Obtains the message identified by mid
@GET
@Path("/{mid}")
@Produces(MediaType.APPLICATION_JSON)
public Message getMessage(@PathParam("mid") long mid);

/**
 * Ob
 * @pa
 * @re                                              exists,
 */
@GET
@Path("/{mid}/body")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public byte[] getMessageBody(@PathParam("mid") long mid);
```

Since both methods return a value to the client, both need to be annotated with @Produces.
Typically, methods that return byte information (i.e., byte[]) produce APPLICATION_OCTET_STREAM. While methods that return Java classes or native types produce APPLICATION_JSON.
The annotation defines how data is serialized in the HTTP reply.

DEFINING THE SERVICE INTERFACE

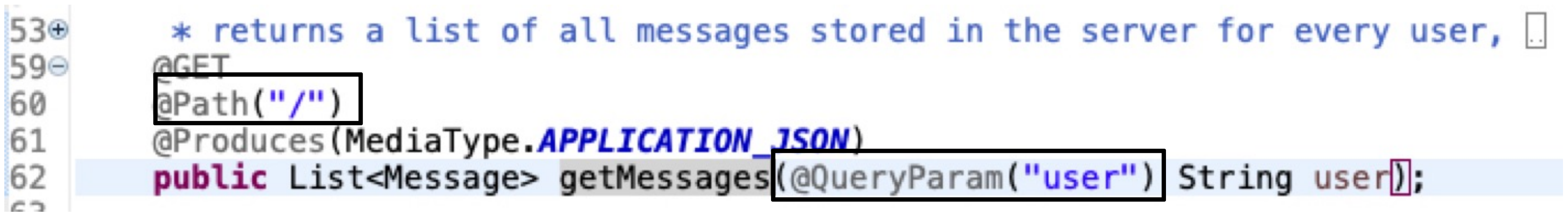
The final method will allow to list existing messages in the server.
It has an optional parameter that if provided, only lists the messages in the inbox of a single user (given his username)



```java
     * returns a list of all messages stored in the server for every user, 
    @GET
    @Path("/")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Message> getMessages(@QueryParam("user") String user);
```

DEFINING THE SERVICE INTERFACE

The final method will allow to list existing messages in the server.
It has an optional parameter that if provided, only lists the messages in the inbox of a single user (given his username)



```
     * returns a list of all messages stored in the server for every user, 
    @GET
    @Path("/")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Message> getMessages(@QueryParam("user") String user);
```

The optional parameter uses the @QueryParam with an argument that is the name of the optional parameter.

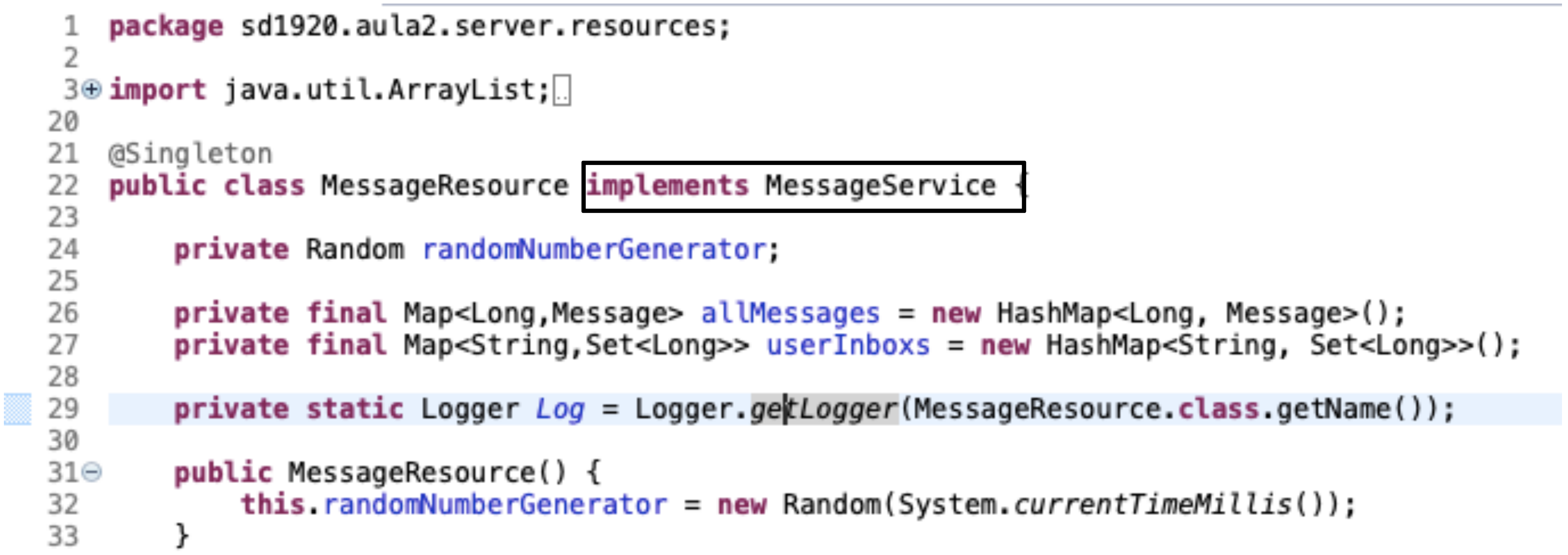
Note that query parameters are not part of the @Path (otherwise they would be mandatory)

They are however passed in the URL using the ? character. E.g., http://myserver:8080/rest/messages?user=jleitao

More about Annotations and Methods

- GET and DELETE are similar.
  - They should avoid to send information in the body of the request (and hence usually do not have a @Consumes Annotation).
- POST and PUT are similar.
  - They should always send a representation of the resource being manipulated in the body of the HTTP request (and hence usually have a @Consumes Annotation).
- GET should always return a representation of a resource.
  - (Therefore, a @Produces Annotation is frequent).

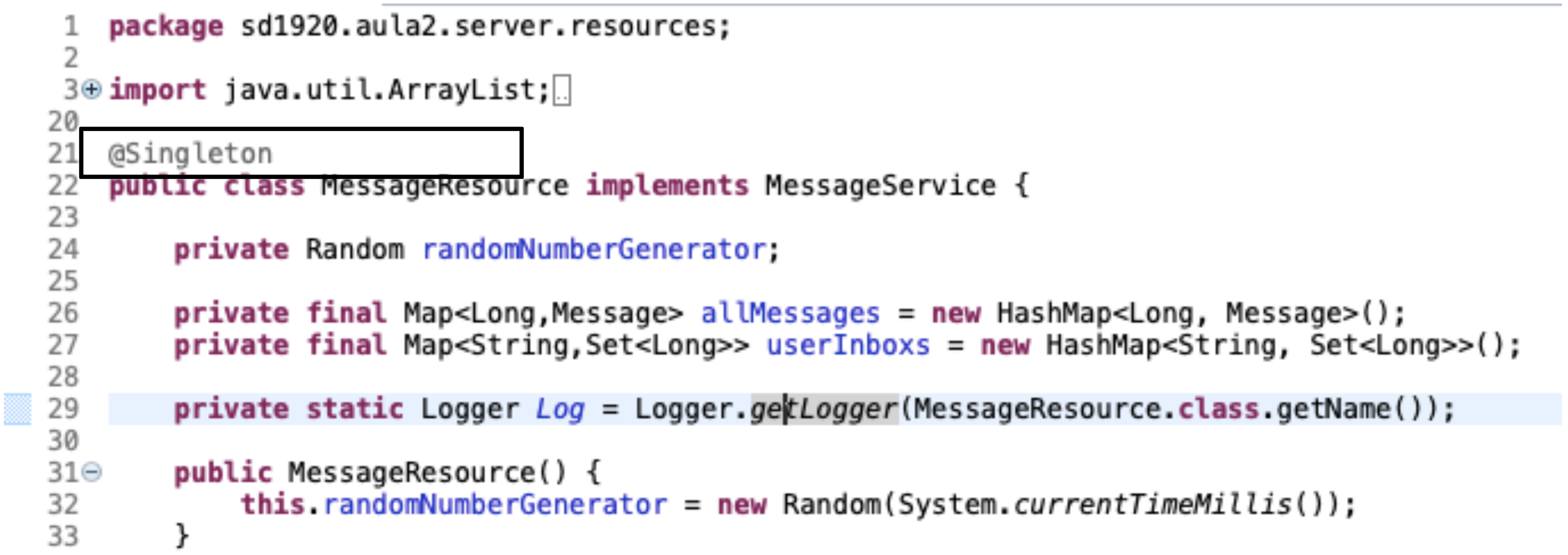
IMPLEMENTING THE SERVICE



```java
package sd1920.aula2.server.resources;

import java.util.ArrayList;

@Singleton
public class MessageResource implements MessageService {

    private Random randomNumberGenerator;

    private final Map<Long,Message> allMessages = new HashMap<Long, Message>();
    private final Map<String,Set<Long>> userInboxs = new HashMap<String, Set<Long>>();

    private static Logger Log = Logger.getLogger(MessageResource.class.getName());

    public MessageResource() {
        this.randomNumberGenerator = new Random(System.currentTimeMillis());
    }
```

Regular Java Class that implements the Interface with the annotations
(The annotations are associated to the class and methods through inheritance)

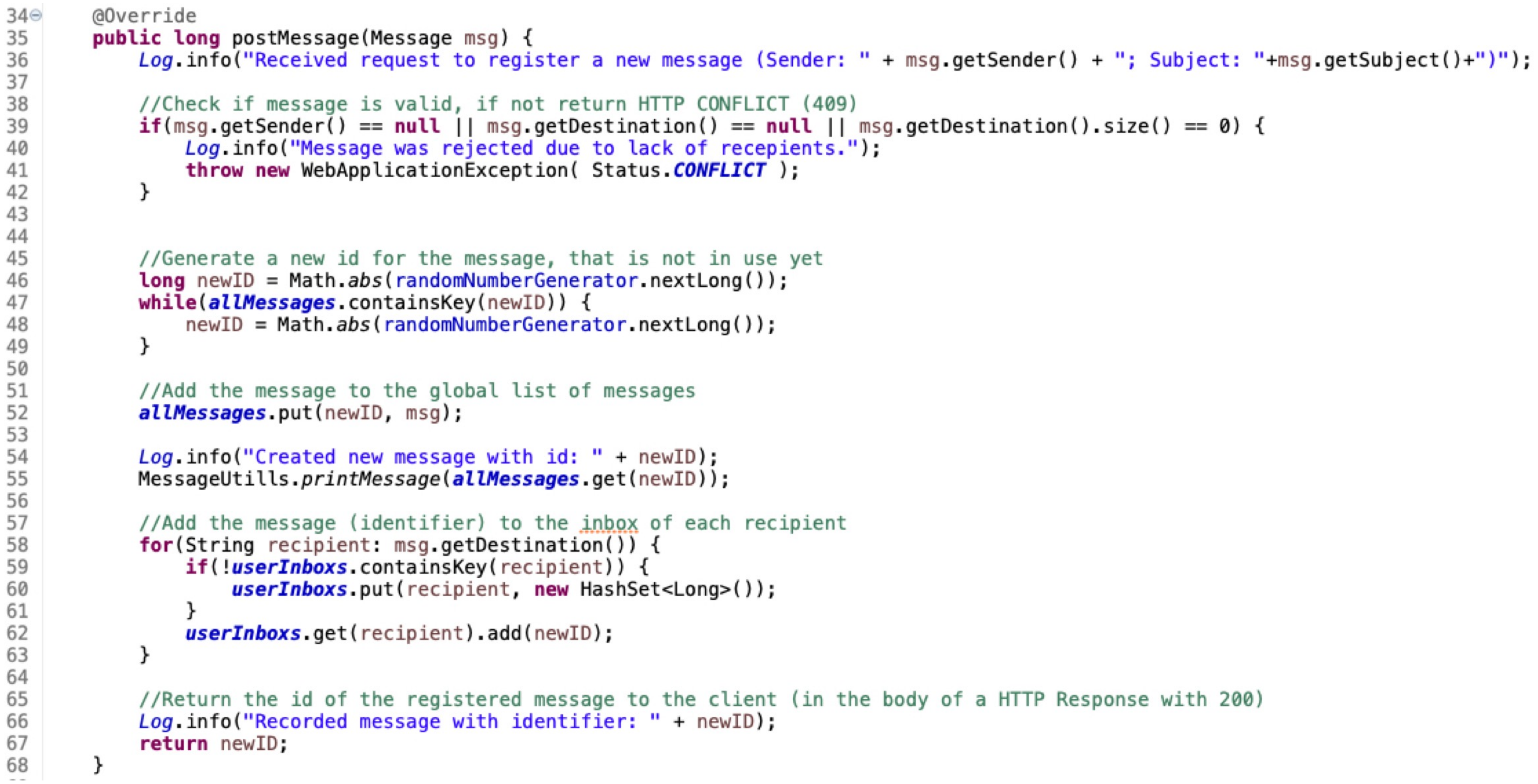
IMPLEMENTING THE SERVICE



```java
package sd1920.aula2.server.resources;

import java.util.ArrayList;

@Singleton
public class MessageResource implements MessageService {

    private Random randomNumberGenerator;

    private final Map<Long,Message> allMessages = new HashMap<Long, Message>();
    private final Map<String,Set<Long>> userInboxs = new HashMap<String, Set<Long>>();

    private static Logger Log = Logger.getLogger(MessageResource.class.getName());

    public MessageResource() {
        this.randomNumberGenerator = new Random(System.currentTimeMillis());
    }
```

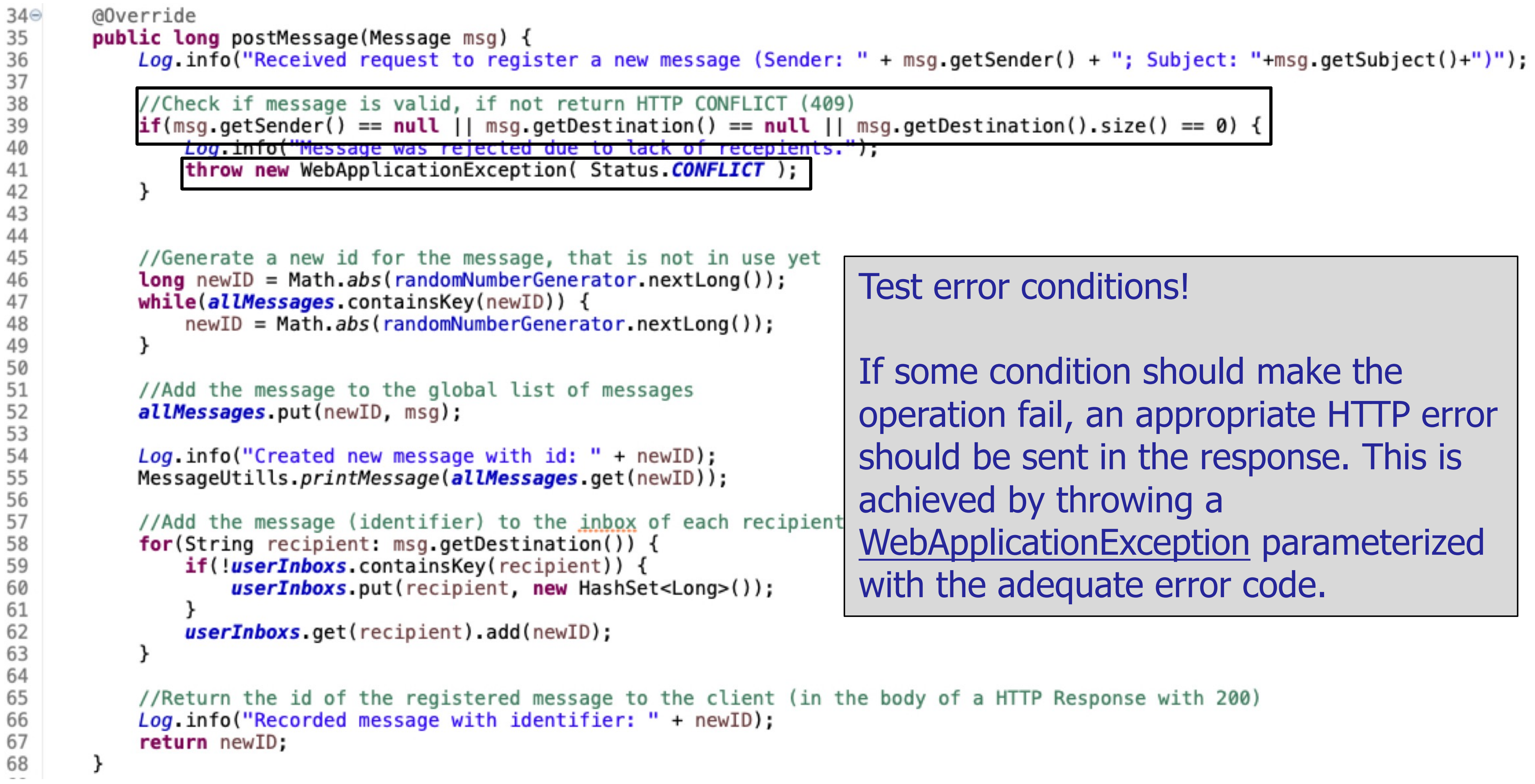
Resources that have internal state should be defined as **@Singleton**, so that a single instance exists in the server. Otherwise, the server will create an instance per request.

IMPLEMENTING THE SERVICE: POST MESSAGE



```java
@Override
public long postMessage(Message msg) {
    Log.info("Received request to register a new message (Sender: " + msg.getSender() + "; Subject: "+msg.getSubject()+")");

    //Check if message is valid, if not return HTTP CONFLICT (409)
    if(msg.getSender() == null || msg.getDestination() == null || msg.getDestination().size() == 0) {
        Log.info("Message was rejected due to lack of recepients.");
        throw new WebApplicationException( Status.CONFLICT );
    }


    //Generate a new id for the message, that is not in use yet
    long newID = Math.abs(randomNumberGenerator.nextLong());
    while(allMessages.containsKey(newID)) {
        newID = Math.abs(randomNumberGenerator.nextLong());
    }

    //Add the message to the global list of messages
    allMessages.put(newID, msg);

    Log.info("Created new message with id: " + newID);
    MessageUtills.printMessage(allMessages.get(newID));

    //Add the message (identifier) to the inbox of each recipient
    for(String recipient: msg.getDestination()) {
        if(!userInboxs.containsKey(recipient)) {
            userInboxs.put(recipient, new HashSet<Long>());
        }
        userInboxs.get(recipient).add(newID);
    }

    //Return the id of the registered message to the client (in the body of a HTTP Response with 200)
    Log.info("Recorded message with identifier: " + newID);
    return newID;
}
```

IMPLEMENTING THE SERVICE: POST MESSAGE



```java
    @Override
    public long postMessage(Message msg) {
        Log.info("Received request to register a new message (Sender: " + msg.getSender() + "; Subject: "+msg.getSubject()+")");

        //Check if message is valid, if not return HTTP CONFLICT (409)
        if(msg.getSender() == null || msg.getDestination() == null || msg.getDestination().size() == 0) {
            Log.info("Message was rejected due to lack of recepients.");
            throw new WebApplicationException( Status.CONFLICT );
        }


        //Generate a new id for the message, that is not in use yet
        long newID = Math.abs(randomNumberGenerator.nextLong());
        while(allMessages.containsKey(newID)) {
            newID = Math.abs(randomNumberGenerator.nextLong());
        }

        //Add the message to the global list of messages
        allMessages.put(newID, msg);

        Log.info("Created new message with id: " + newID);
        MessageUtills.printMessage(allMessages.get(newID));

        //Add the message (identifier) to the inbox of each recipient
        for(String recipient: msg.getDestination()) {
            if(!userInboxs.containsKey(recipient)) {
                userInboxs.put(recipient, new HashSet<Long>());
            }
            userInboxs.get(recipient).add(newID);
        }

        //Return the id of the registered message to the client (in the body of a HTTP Response with 200)
        Log.info("Recorded message with identifier: " + newID);
        return newID;
    }
```

Test error conditions!

If some condition should make the operation fail, an appropriate HTTP error should be sent in the response. This is achieved by throwing a WebApplicationException parameterized with the adequate error code.

Important HTTP Response Codes

Range 100 – 199: Information (rarely seen)

Range 200 – 299: Success

- 200: OK (the operation was successful, and the reply contains information)
- 204: No Content (the operation was successful but there is no information returned).

Range 300 – 399: Redirection: additional action is required

- 301: Moved Permanently (the resource is now represented by a new URL, which is provided in this answer)

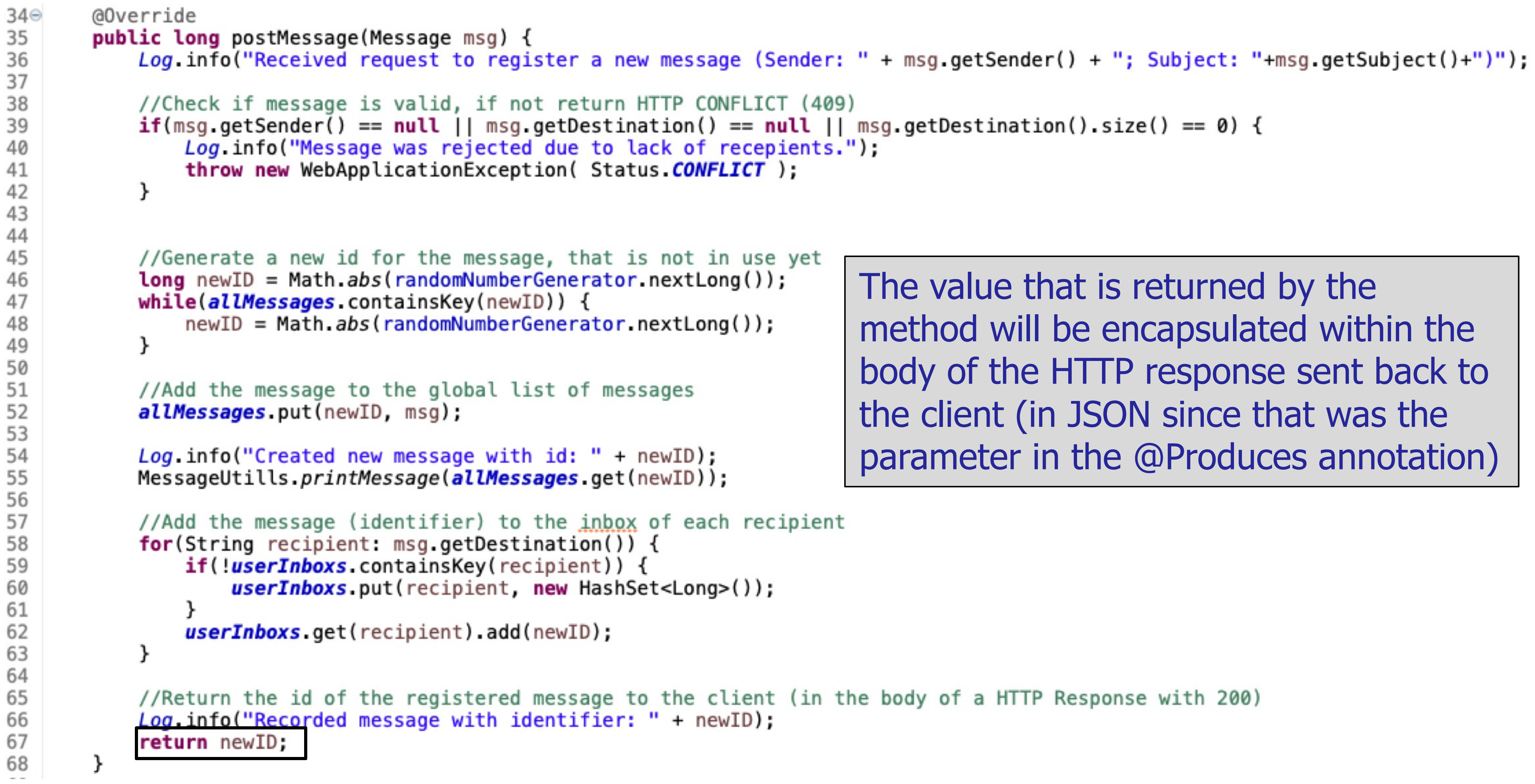
Range 400 – 499: Client Error (e.g., preparing request)

- 404: Page/Resource not found
- 409: Conflict – executing the request violates logic rules

Range 500 – 599: Server Error

- 500: Internal Server Error – usually means an unhandled exception was thrown while executing request

IMPLEMENTING THE SERVICE: POST MESSAGE



```java
    @Override
    public long postMessage(Message msg) {
        Log.info("Received request to register a new message (Sender: " + msg.getSender() + "; Subject: "+msg.getSubject()+")");

        //Check if message is valid, if not return HTTP CONFLICT (409)
        if(msg.getSender() == null || msg.getDestination() == null || msg.getDestination().size() == 0) {
            Log.info("Message was rejected due to lack of recepients.");
            throw new WebApplicationException( Status.CONFLICT );
        }


        //Generate a new id for the message, that is not in use yet
        long newID = Math.abs(randomNumberGenerator.nextLong());
        while(allMessages.containsKey(newID)) {
            newID = Math.abs(randomNumberGenerator.nextLong());
        }

        //Add the message to the global list of messages
        allMessages.put(newID, msg);

        Log.info("Created new message with id: " + newID);
        MessageUtills.printMessage(allMessages.get(newID));

        //Add the message (identifier) to the inbox of each recipient
        for(String recipient: msg.getDestination()) {
            if(!userInboxs.containsKey(recipient)) {
                userInboxs.put(recipient, new HashSet<Long>());
            }
            userInboxs.get(recipient).add(newID);
        }

        //Return the id of the registered message to the client (in the body of a HTTP Response with 200)
        Log.info("Recorded message with identifier: " + newID);
        return newID;
    }
```

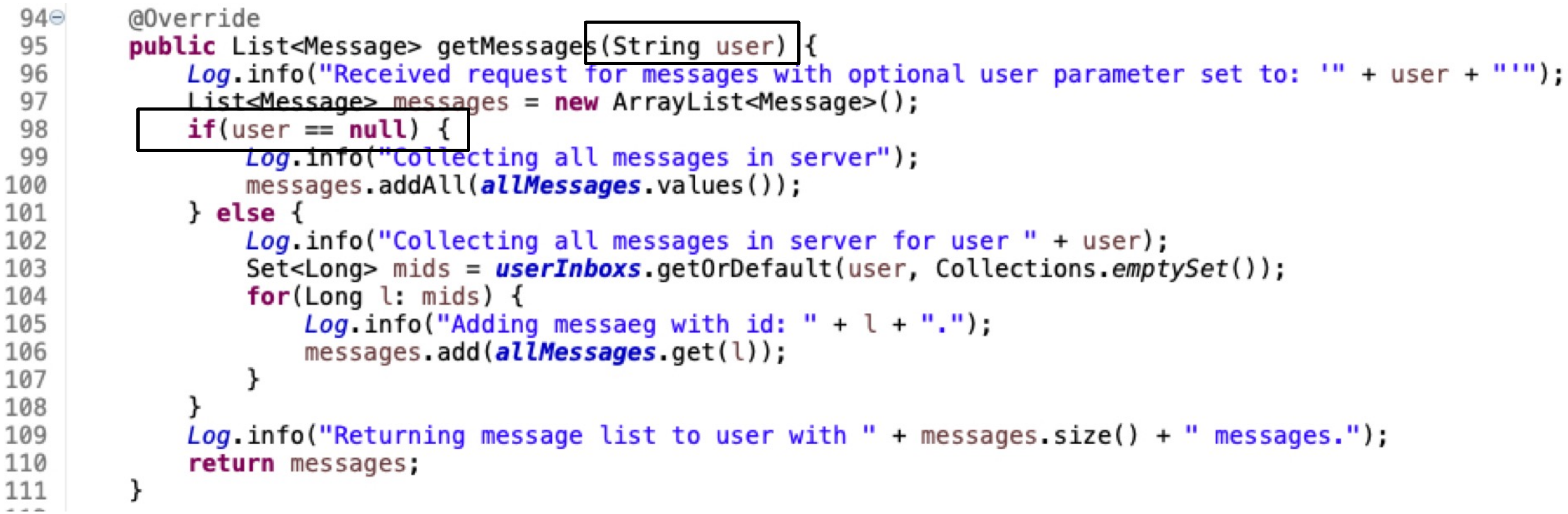
The value that is returned by the method will be encapsulated within the body of the HTTP response sent back to the client (in JSON since that was the parameter in the @Produces annotation)

IMPLEMENTING THE SERVICE: GET MESSAGE

Test error condition and return 404 if the operation targets a message that does not exists.

```
@Override
public Message getMessage(long mid) {
    Log.info("Received request for message with id: " + mid +".");
    if(!allMessages.containsKey(mid)) { //check if message exists
        Log.info("Requested message does not exists.");
        throw new WebApplicationException( Status.NOT_FOUND ); //if not send HTTP 404 back to client
    }

    Log.info("Returning requested message to user.");
    return allMessages.get(mid); //Return message to the client with code HTTP 200
}
```

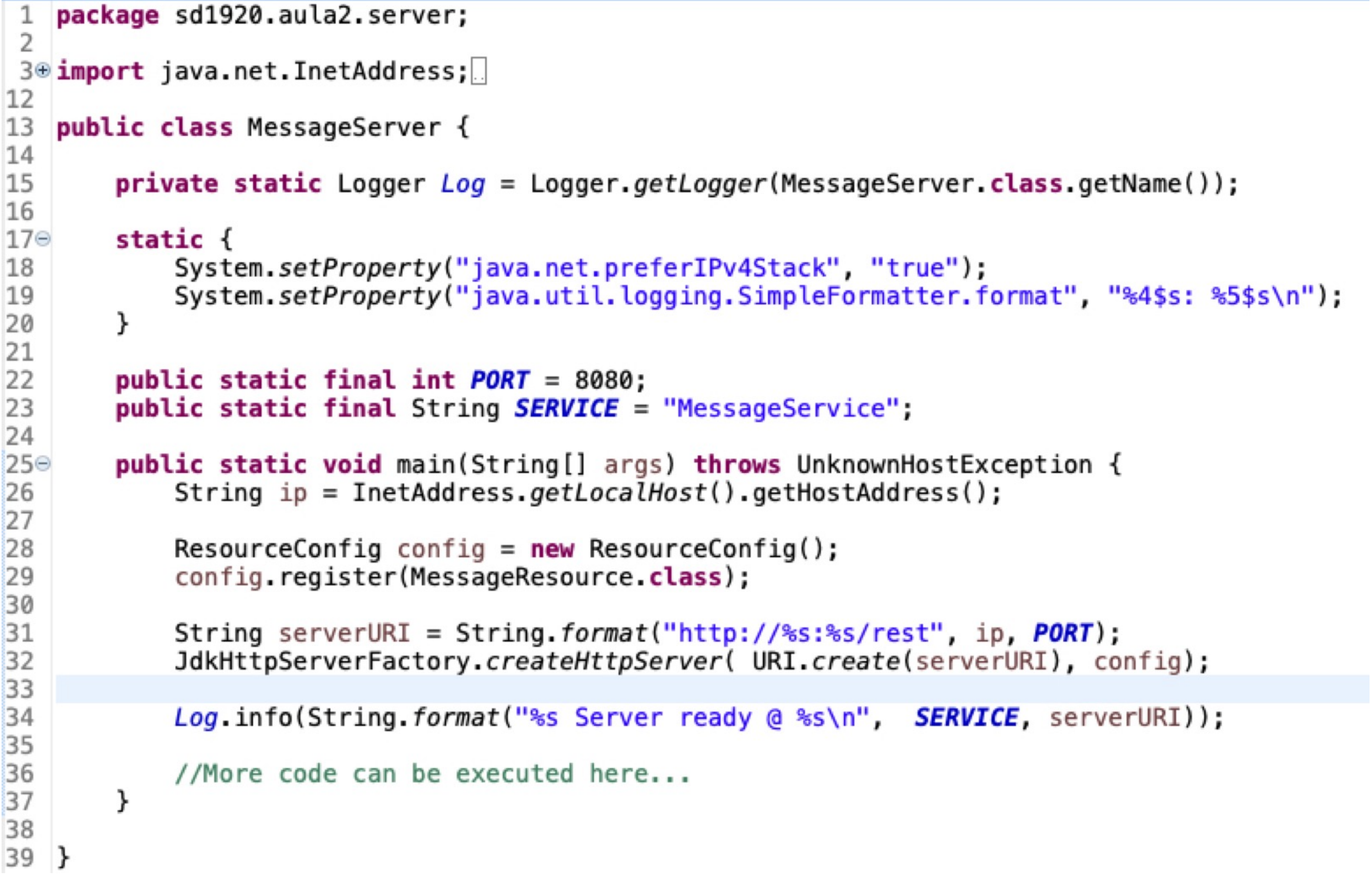
IMPLEMENTING THE SERVICE: GET MESSAGES



```java
    @Override
    public List<Message> getMessages(String user) {
        Log.info("Received request for messages with optional user parameter set to: '" + user + "'");
        List<Message> messages = new ArrayList<Message>();
        if(user == null) {
            Log.info("Collecting all messages in server");
            messages.addAll(allMessages.values());
        } else {
            Log.info("Collecting all messages in server for user " + user);
            Set<Long> mids = userInboxs.getOrDefault(user, Collections.emptySet());
            for(Long l: mids) {
                Log.info("Adding messaeg with id: " + l + ".");
                messages.add(allMessages.get(l));
            }
        }
        Log.info("Returning message list to user with " + messages.size() + " messages.");
        return messages;
    }
```

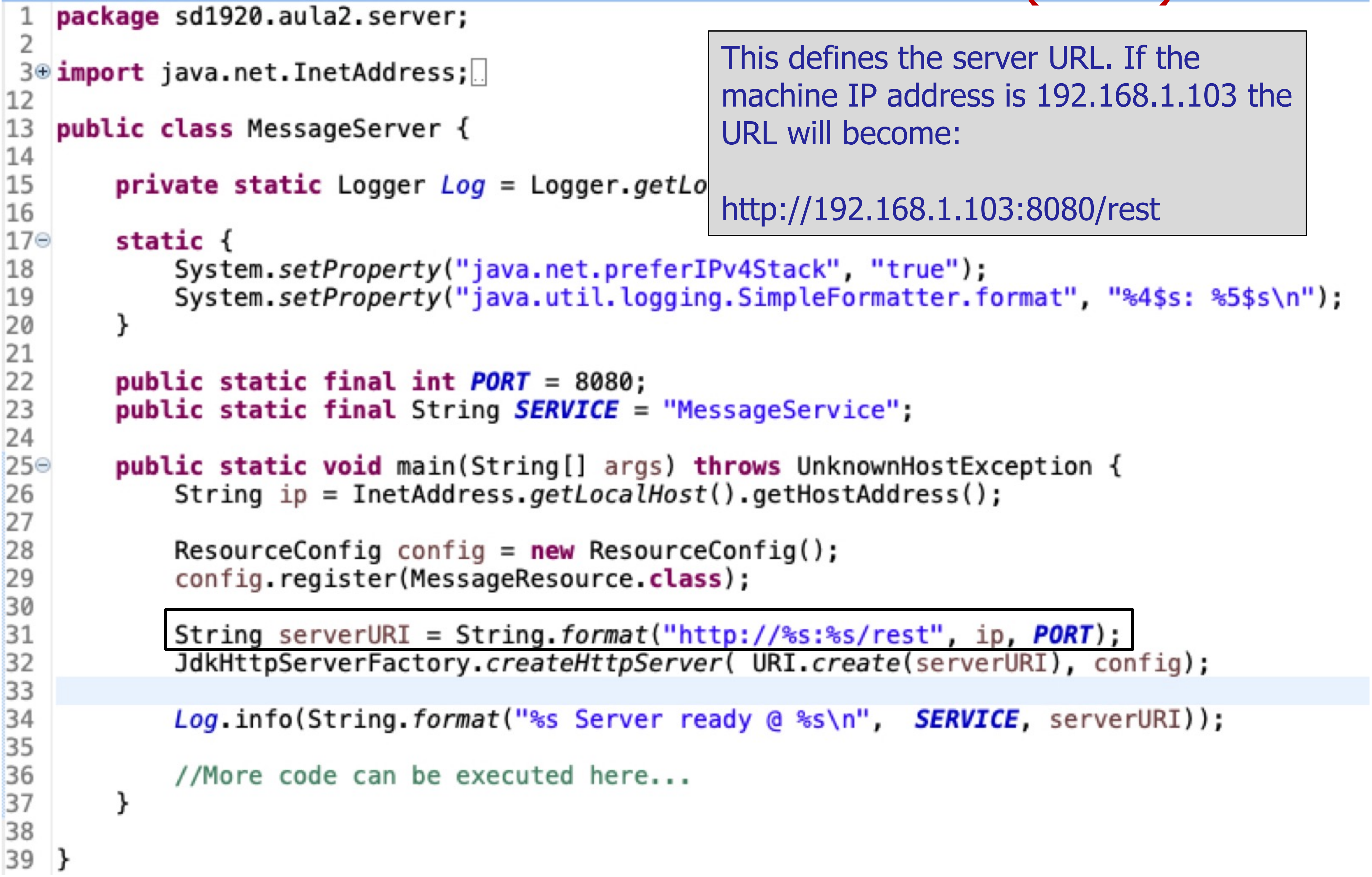
This parameter is obtained from a query param, it is optional and will be null if no value is provided in the request.

Implementing the Service: Server Code (Main)



```java
package sd1920.aula2.server;

import java.net.InetAddress;

public class MessageServer {

    private static Logger Log = Logger.getLogger(MessageServer.class.getName());

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "MessageService";

    public static void main(String[] args) throws UnknownHostException {
        String ip = InetAddress.getLocalHost().getHostAddress();

        ResourceConfig config = new ResourceConfig();
        config.register(MessageResource.class);

        String serverURI = String.format("http://%s:%s/rest", ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
    }

}
```

Implementing the Service: Server Code (Main)

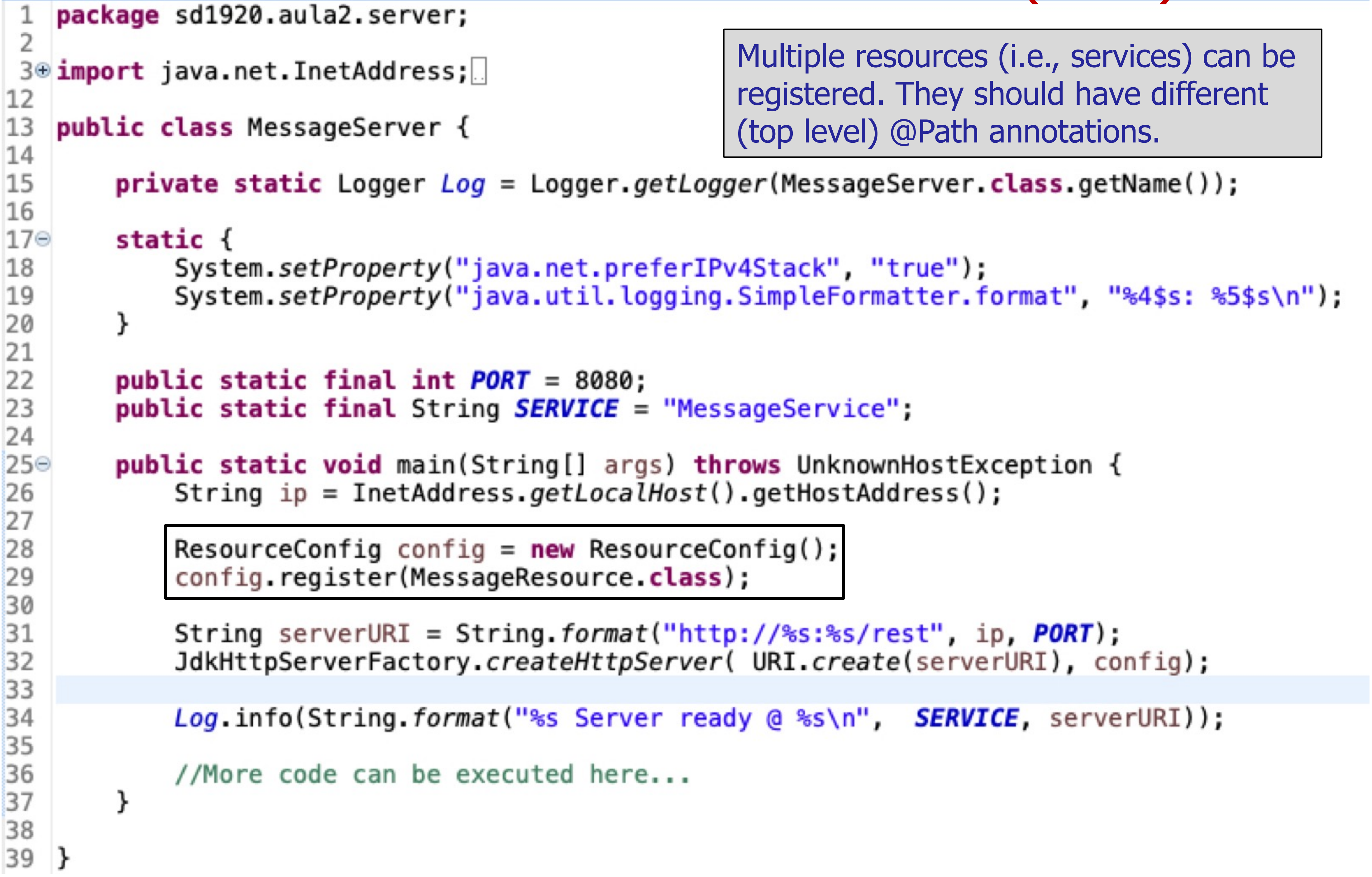


```java
package sd1920.aula2.server;

import java.net.InetAddress;

public class MessageServer {

    private static Logger Log = Logger.getLo

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "MessageService";

    public static void main(String[] args) throws UnknownHostException {
        String ip = InetAddress.getLocalHost().getHostAddress();

        ResourceConfig config = new ResourceConfig();
        config.register(MessageResource.class);

        String serverURI = String.format("http://%s:%s/rest", ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
    }

}
```

This defines the server URL. If the machine IP address is 192.168.1.103 the URL will become:

http://192.168.1.103:8080/rest

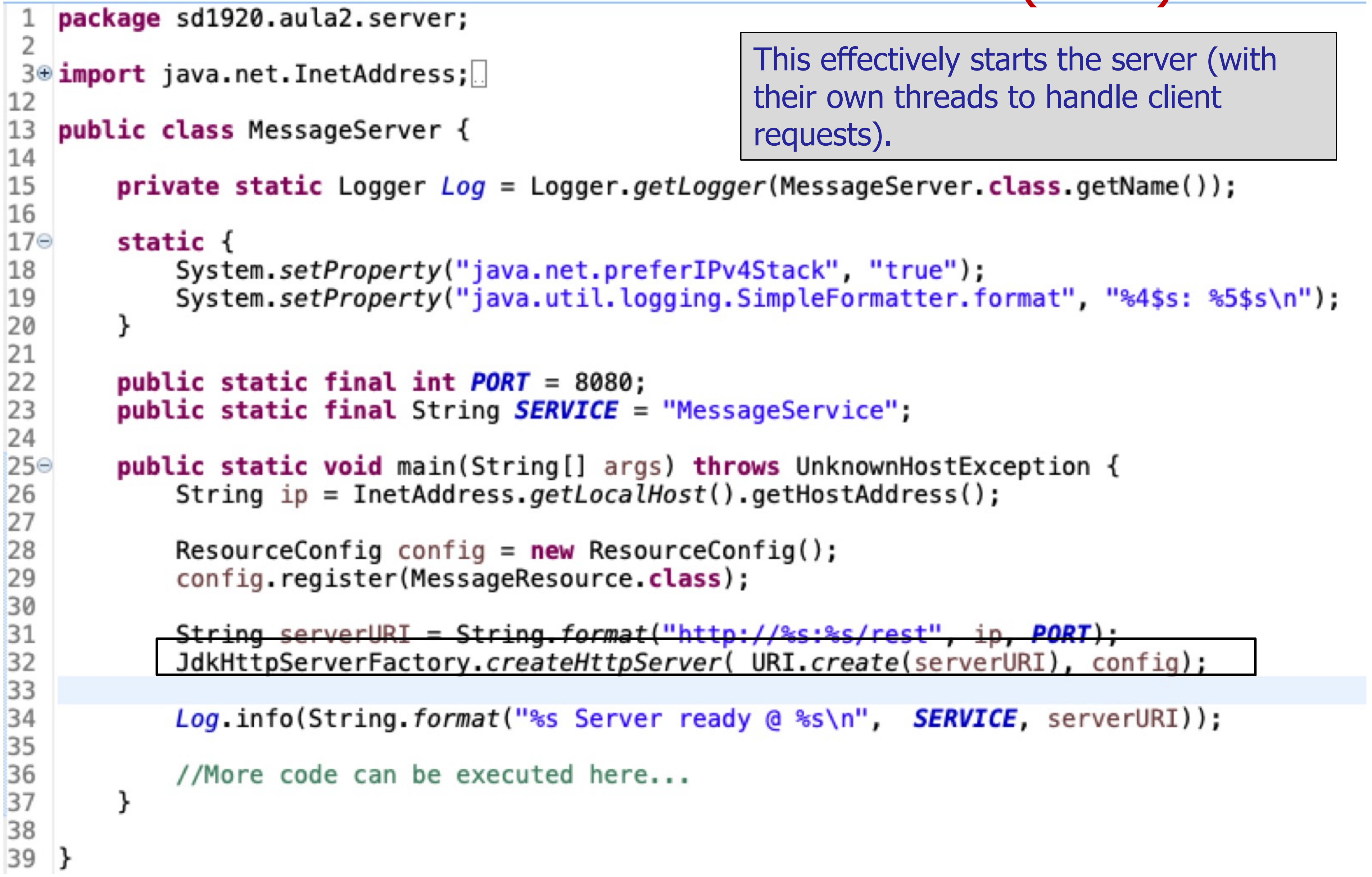
Implementing the Service: Server Code (Main)



```java
package sd1920.aula2.server;

import java.net.InetAddress;

public class MessageServer {

    private static Logger Log = Logger.getLogger(MessageServer.class.getName());

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "MessageService";

    public static void main(String[] args) throws UnknownHostException {
        String ip = InetAddress.getLocalHost().getHostAddress();

        ResourceConfig config = new ResourceConfig();
        config.register(MessageResource.class);

        String serverURI = String.format("http://%s:%s/rest", ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
    }

}
```

Multiple resources (i.e., services) can be registered. They should have different (top level) @Path annotations.

Implementing the Service: Server Code (Main)



```java
package sd1920.aula2.server;

import java.net.InetAddress;

public class MessageServer {

    private static Logger Log = Logger.getLogger(MessageServer.class.getName());

    static {
        System.setProperty("java.net.preferIPv4Stack", "true");
        System.setProperty("java.util.logging.SimpleFormatter.format", "%4$s: %5$s\n");
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "MessageService";

    public static void main(String[] args) throws UnknownHostException {
        String ip = InetAddress.getLocalHost().getHostAddress();

        ResourceConfig config = new ResourceConfig();
        config.register(MessageResource.class);

        String serverURI = String.format("http://%s:%s/rest", ip, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config);

        Log.info(String.format("%s Server ready @ %s\n",  SERVICE, serverURI));

        //More code can be executed here...
    }

}
```

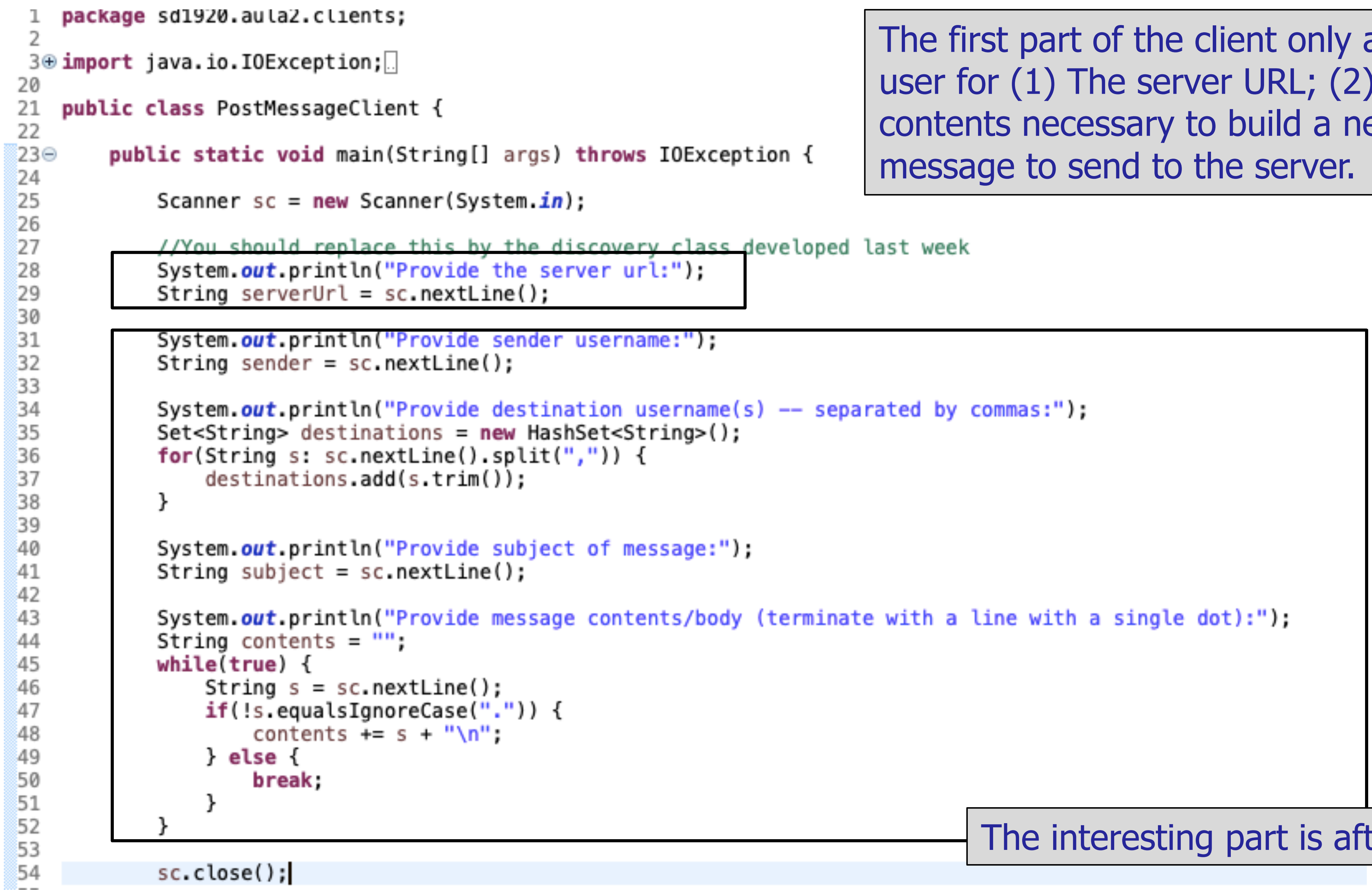
This effectively starts the server (with their own threads to handle client requests).

GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- **Know how to develop a REST Client in Java (using JAX-RX)**
- Use Docker to test your service using your clients

IMPLEMENTING THE CLIENT: POST MESSAGE (1/2)

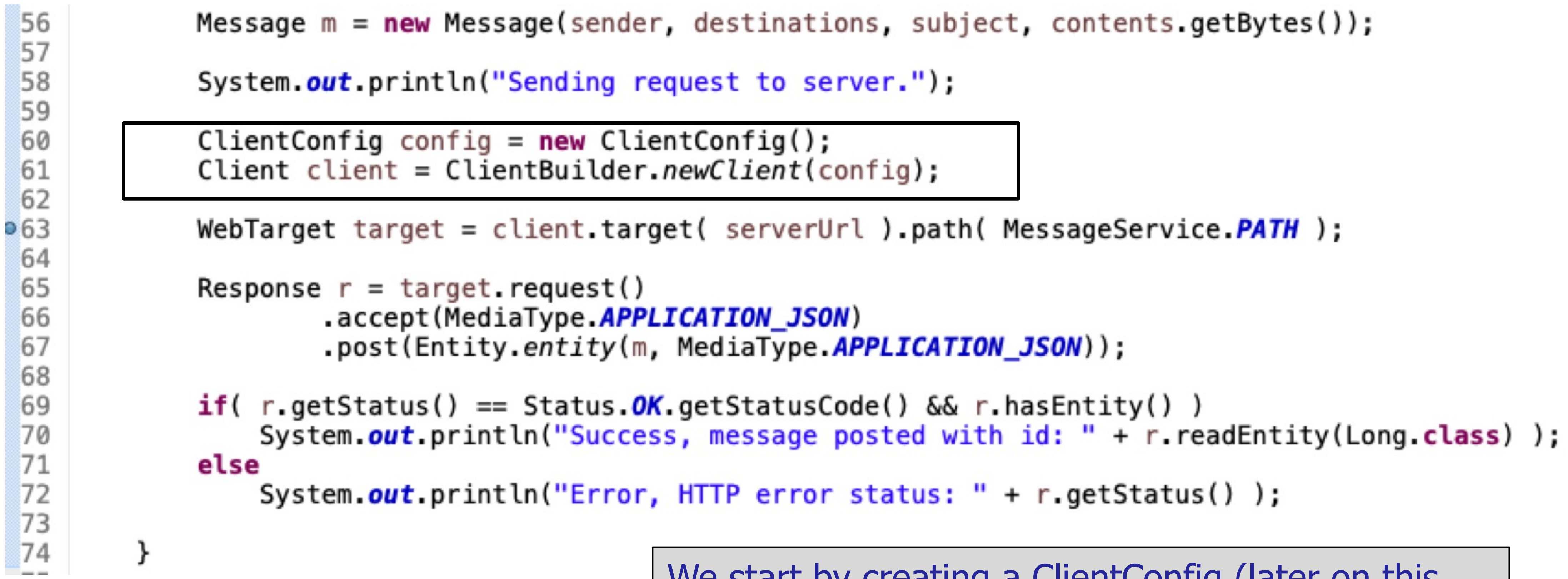


```java
package sd1920.aula2.clients;

import java.io.IOException;

public class PostMessageClient {

    public static void main(String[] args) throws IOException {

        Scanner sc = new Scanner(System.in);

        //You should replace this by the discovery class developed last week
        System.out.println("Provide the server url:");
        String serverUrl = sc.nextLine();

        System.out.println("Provide sender username:");
        String sender = sc.nextLine();

        System.out.println("Provide destination username(s) -- separated by commas:");
        Set<String> destinations = new HashSet<String>();
        for(String s: sc.nextLine().split(",")) {
            destinations.add(s.trim());
        }

        System.out.println("Provide subject of message:");
        String subject = sc.nextLine();

        System.out.println("Provide message contents/body (terminate with a line with a single dot):");
        String contents = "";
        while(true) {
            String s = sc.nextLine();
            if(!s.equalsIgnoreCase(".")) {
                contents += s + "\n";
            } else {
                break;
            }
        }

        sc.close();
```

The first part of the client only asks the user for (1) The server URL; (2) the contents necessary to build a new message to send to the server.

The interesting part is after this

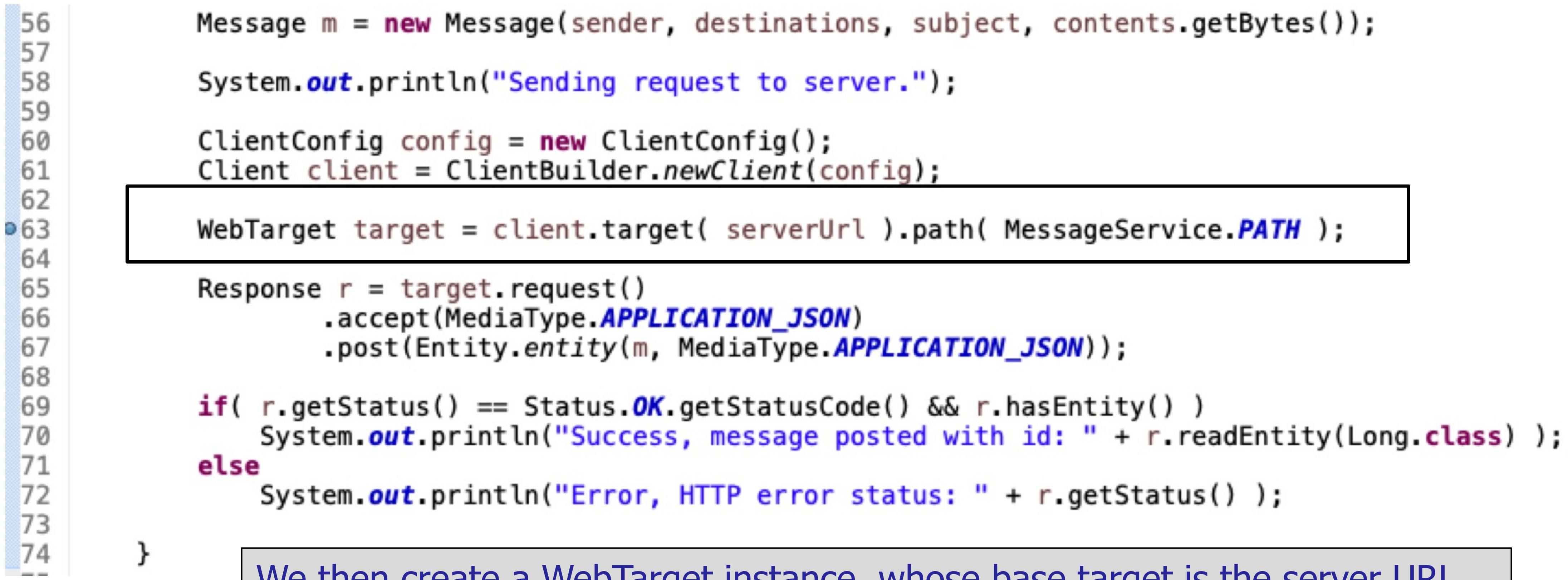
IMPLEMENTING THE CLIENT: POST MESSAGE (2/2)



```java
        Message m = new Message(sender, destinations, subject, contents.getBytes());

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

        Response r = target.request()
                .accept(MediaType.APPLICATION_JSON)
                .post(Entity.entity(m, MediaType.APPLICATION_JSON));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Success, message posted with id: " + r.readEntity(Long.class) );
        else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }
```

We start by creating a ClientConfig (later on this can be used to control the behavior of the client) and from that generate an instance of a Client.

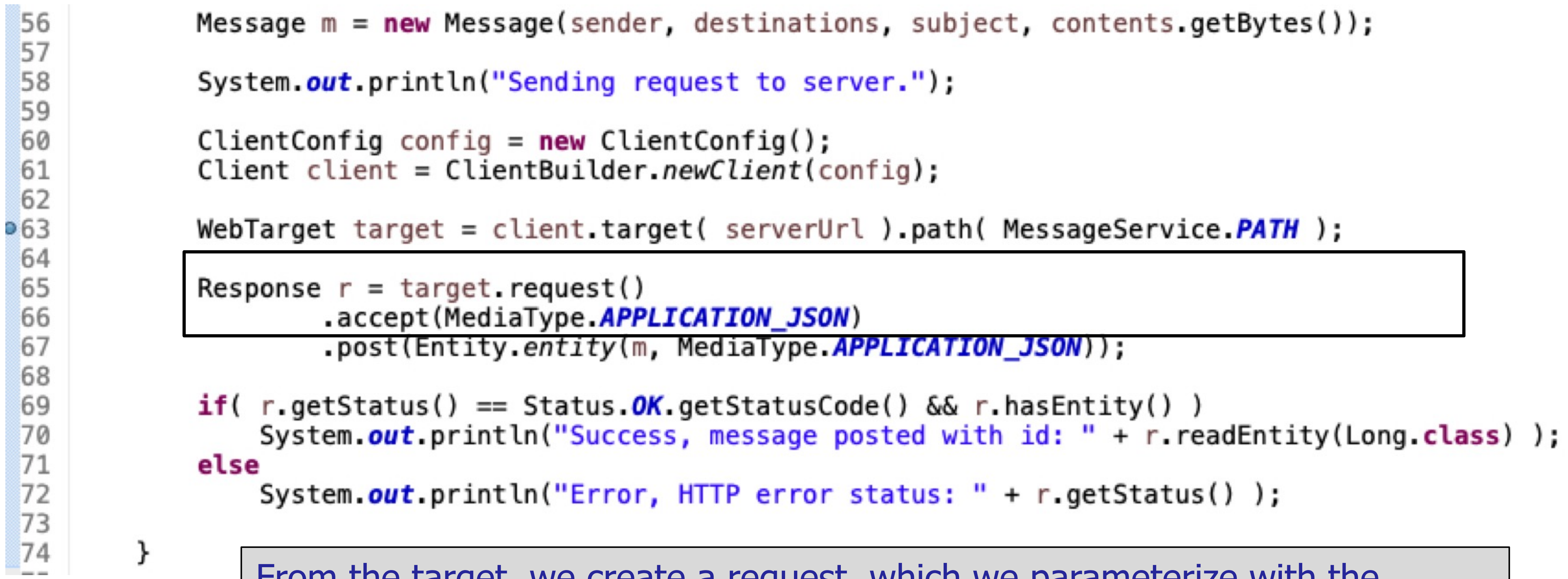
IMPLEMENTING THE CLIENT: POST MESSAGE (2/2)



```
        Message m = new Message(sender, destinations, subject, contents.getBytes());

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

        Response r = target.request()
                .accept(MediaType.APPLICATION_JSON)
                .post(Entity.entity(m, MediaType.APPLICATION_JSON));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Success, message posted with id: " + r.readEntity(Long.class) );
        else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }
```

We then create a WebTarget instance, whose base target is the server URL (e.g., http://192.168.1.103:8080/rest). We then can concatenate any number of other elements to the URL. Here we are just adding the path corresponding to the service (enclosed in the top level @Path annotation of the service): e.g., http://192.168.1.103:8080/rest/messages

Implementing the Client: Post Message (2/2)

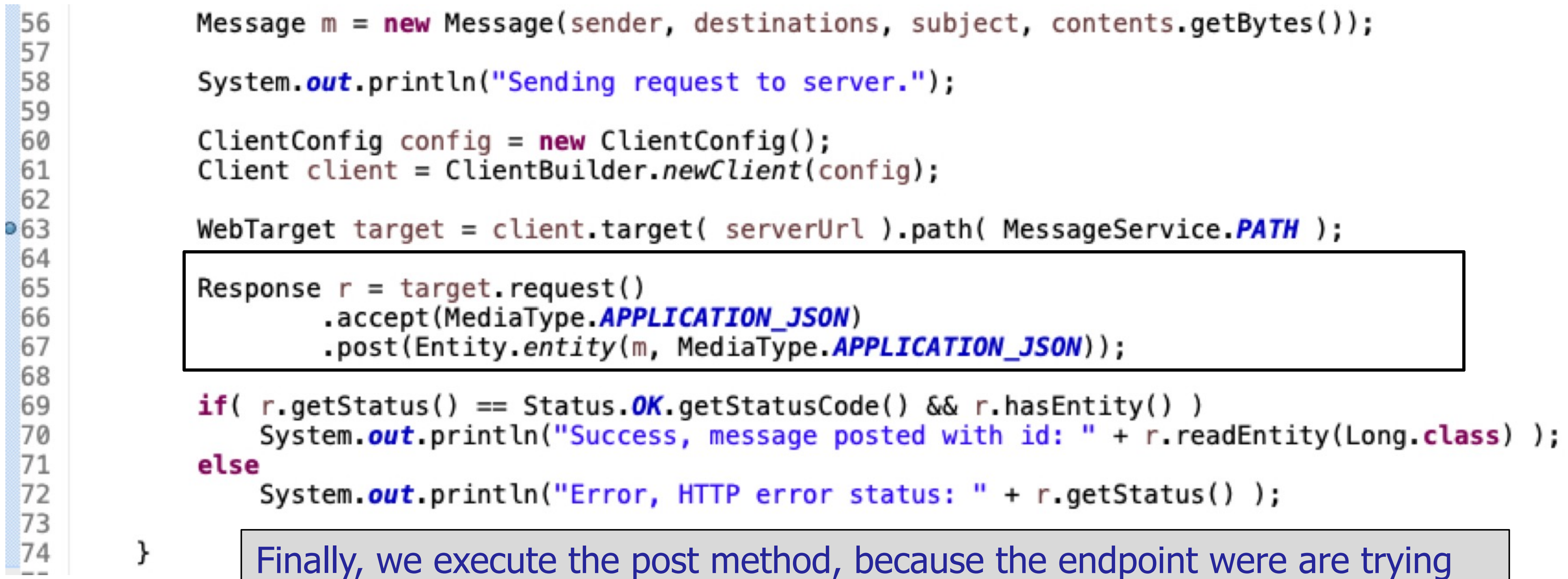


```java
        Message m = new Message(sender, destinations, subject, contents.getBytes());

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

        Response r = target.request()
                .accept(MediaType.APPLICATION_JSON)
                .post(Entity.entity(m, MediaType.APPLICATION_JSON));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Success, message posted with id: " + r.readEntity(Long.class) );
        else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }
```

From the target, we create a request, which we parameterize with the .accept() method to state what is the format in which we can receive the return value in the body of the HTTP response (must match the @Produces annotation on the server).

This is optional and is only performed when the endpoint returns some value.

Implementing the Client: Post Message (2/2)

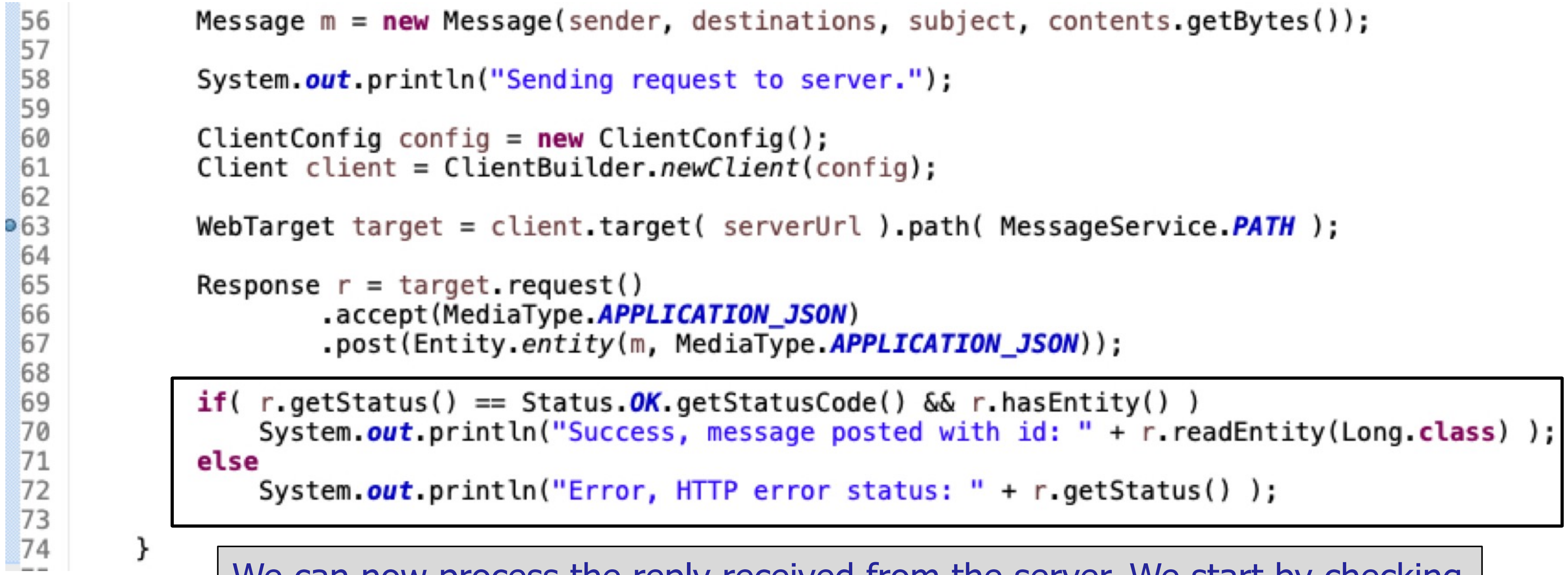


```java
        Message m = new Message(sender, destinations, subject, contents.getBytes());

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

        Response r = target.request()
                .accept(MediaType.APPLICATION_JSON)
                .post(Entity.entity(m, MediaType.APPLICATION_JSON));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Success, message posted with id: " + r.readEntity(Long.class) );
        else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }
```

Finally, we execute the post method, because the endpoint were are trying to use is a POST HTTP operation. As an argument we can encode the parameter that is passed in the body of the HTTP request using the Entity classe. The second argument must match the annotation @Consumes on the server side. The argument of Post is optional.

The invocation of post effectively executes the request to the server and waits for a response that is returned.

IMPLEMENTING THE CLIENT: POST MESSAGE (2/2)



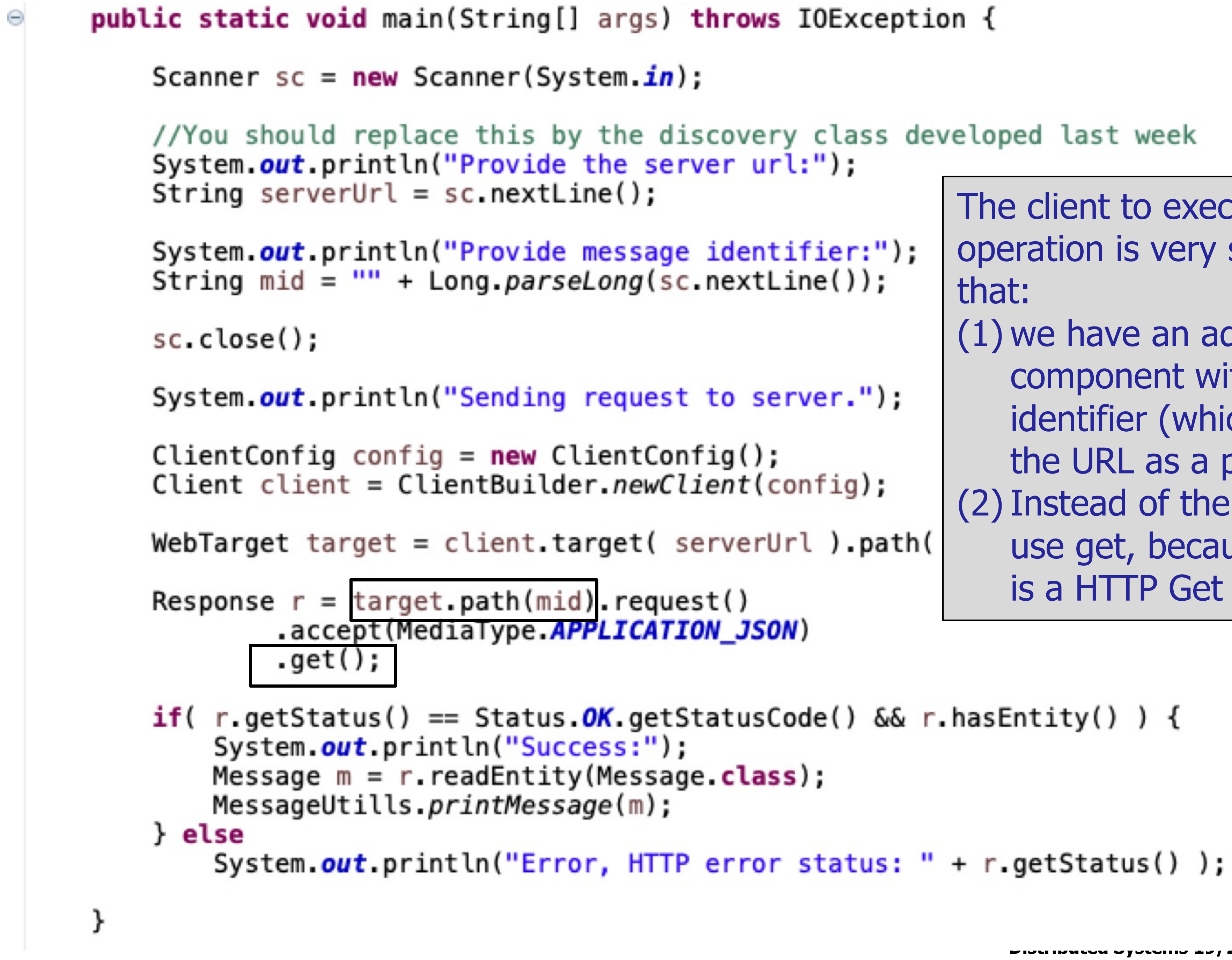
```java
        Message m = new Message(sender, destinations, subject, contents.getBytes());

        System.out.println("Sending request to server.");

        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);

        WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

        Response r = target.request()
                .accept(MediaType.APPLICATION_JSON)
                .post(Entity.entity(m, MediaType.APPLICATION_JSON));

        if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() )
            System.out.println("Success, message posted with id: " + r.readEntity(Long.class) );
        else
            System.out.println("Error, HTTP error status: " + r.getStatus() );

    }
```

We can now process the reply received from the server. We start by checking the HTTP response code (OK – 200), and check if the body of the reply contains an object.

If so, we access the contents of the body with the method readEntity, which is parameterized with the class we want to read from the body.

If the request failed, we print the HTTP response code.

Implementing the Client: Get Message

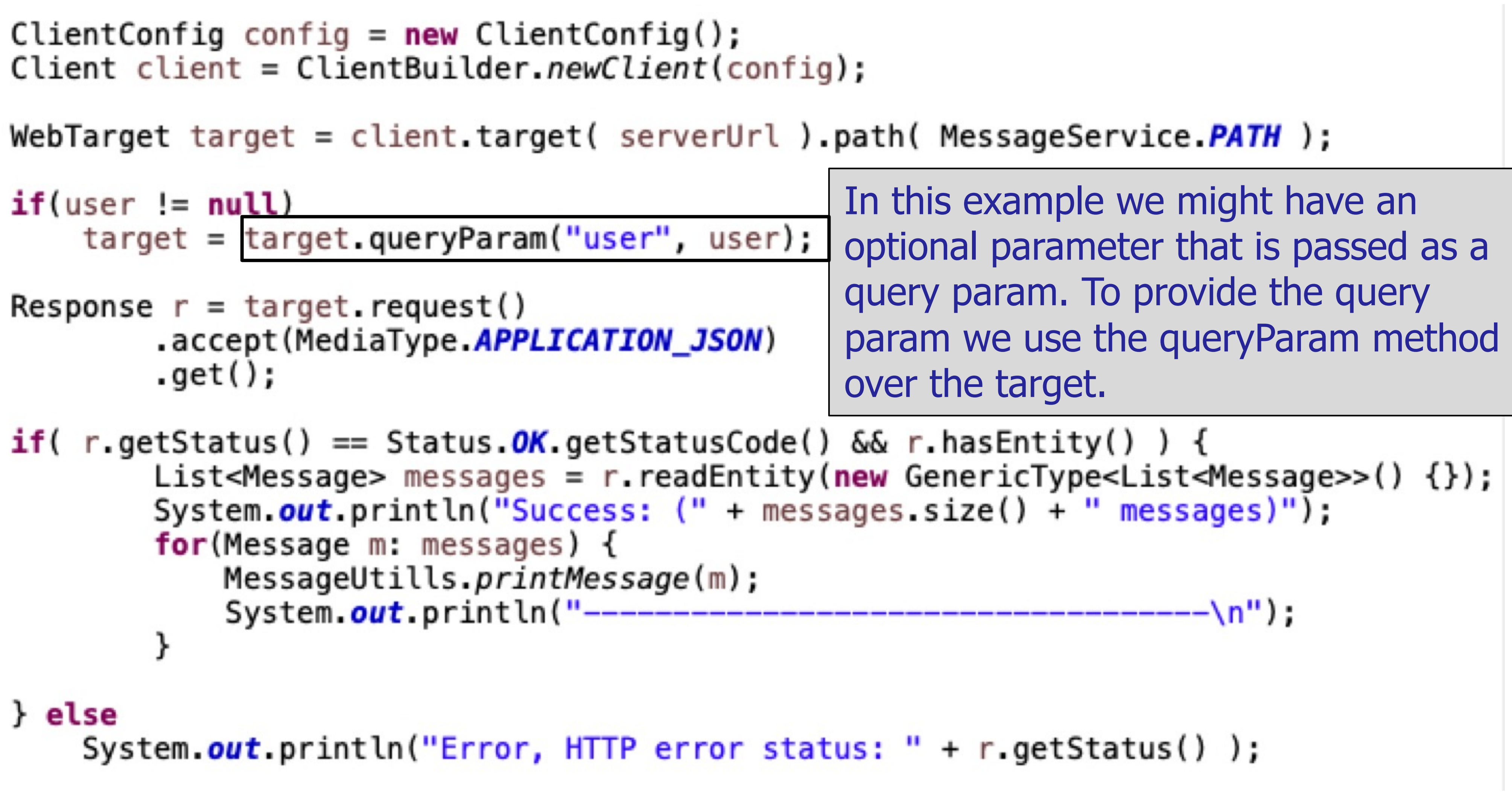


```
public static void main(String[] args) throws IOException {

    Scanner sc = new Scanner(System.in);

    //You should replace this by the discovery class developed last week
    System.out.println("Provide the server url:");
    String serverUrl = sc.nextLine();

    System.out.println("Provide message identifier:");
    String mid = "" + Long.parseLong(sc.nextLine());

    sc.close();

    System.out.println("Sending request to server.");

    ClientConfig config = new ClientConfig();
    Client client = ClientBuilder.newClient(config);

    WebTarget target = client.target( serverUrl ).path(

    Response r = target.path(mid).request()
            .accept(MediaType.APPLICATION_JSON)
            .get();

    if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
        System.out.println("Success:");
        Message m = r.readEntity(Message.class);
        MessageUtills.printMessage(m);
    } else
        System.out.println("Error, HTTP error status: " + r.getStatus() );

}
```

The client to execute the get operation is very similar, except that:

(1) we have an additional path component with the message identifier (which is passed in the URL as a path parameter)
(2) Instead of the method post we use get, because this endpoint is a HTTP Get operation

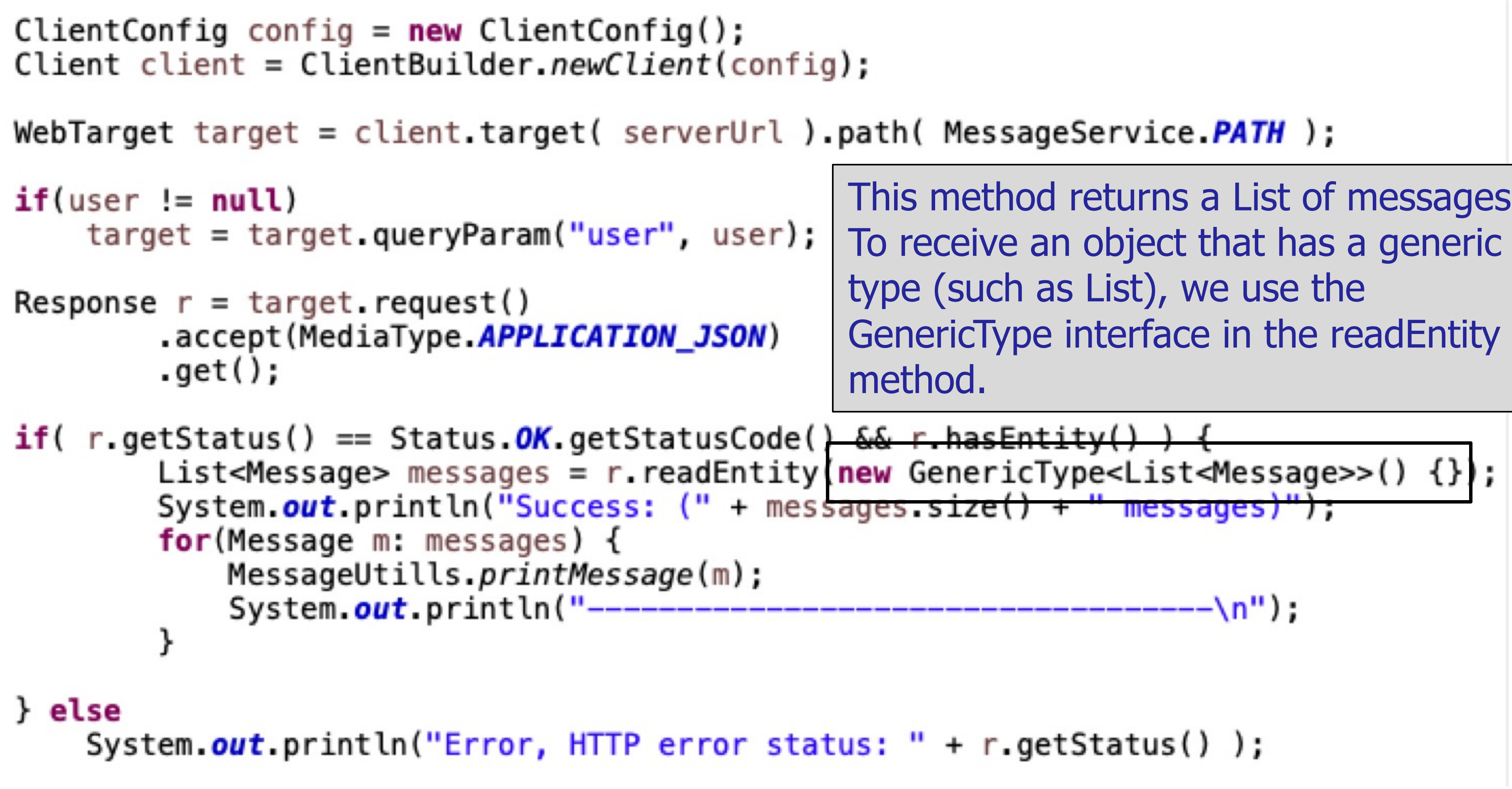
IMPLEMENTING THE CLIENT: GET MESSAGES



```java
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

if(user != null)
    target = target.queryParam("user", user);

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .get();

if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
        List<Message> messages = r.readEntity(new GenericType<List<Message>>() {});
        System.out.println("Success: (" + messages.size() + " messages)");
        for(Message m: messages) {
            MessageUtills.printMessage(m);
            System.out.println("-----------------------------------\n");
        }

} else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

In this example we might have an optional parameter that is passed as a query param. To provide the query param we use the queryParam method over the target.

IMPLEMENTING THE CLIENT: GET MESSAGES



```java
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);

WebTarget target = client.target( serverUrl ).path( MessageService.PATH );

if(user != null)
    target = target.queryParam("user", user);

Response r = target.request()
        .accept(MediaType.APPLICATION_JSON)
        .get();

if( r.getStatus() == Status.OK.getStatusCode() && r.hasEntity() ) {
        List<Message> messages = r.readEntity(new GenericType<List<Message>>() {});
        System.out.println("Success: (" + messages.size() + " messages)");
        for(Message m: messages) {
            MessageUtills.printMessage(m);
            System.out.println("-----------------------------------\n");
        }

} else
    System.out.println("Error, HTTP error status: " + r.getStatus() );
```

This method returns a List of messages. To receive an object that has a generic type (such as List), we use the GenericType interface in the readEntity method.

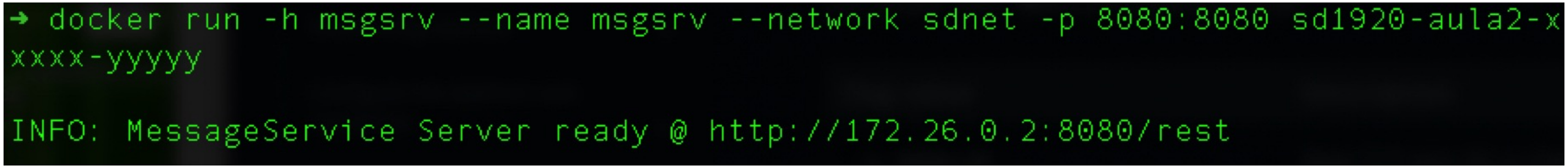
GOALS

In the end of this lab you should be able to:

- Understand what a WebService REST is
- Know how to develop a WS REST and Server in Java (using JAX-RS)
- Know how to develop a REST Client in Java (using JAX-RX)
- **Use Docker to test your service using your clients**

TESTING WITH DOCKER

1. **Build the image (on your project folder run):**
   - mvn clean compile assembly:single docker:build
2. **If you don’t have it yet, create the docker network sdnet**
   - docker network create -d bridge sdnet
3. **Run the server in a named container (with port forwarding)**
   - docker run -h msgsrv --name msgsrv --network sdnet -p 8080:8080 sd1920-aula2-xxxxx-yyyyy

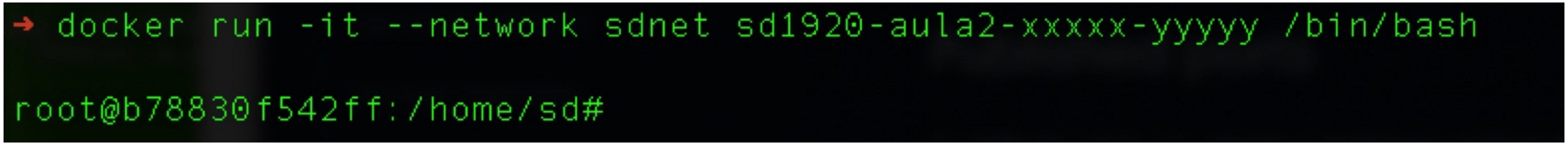


```
➜ docker run -h msgsrv --name msgsrv --network sdnet -p 8080:8080 sd1920-aula2-x
xxxx-yyyyy

INFO: MessageService Server ready @ http://172.26.0.2:8080/rest
```

NOTE: Check the URL for the server – you will need it in the clients.

TESTING WITH DOCKER

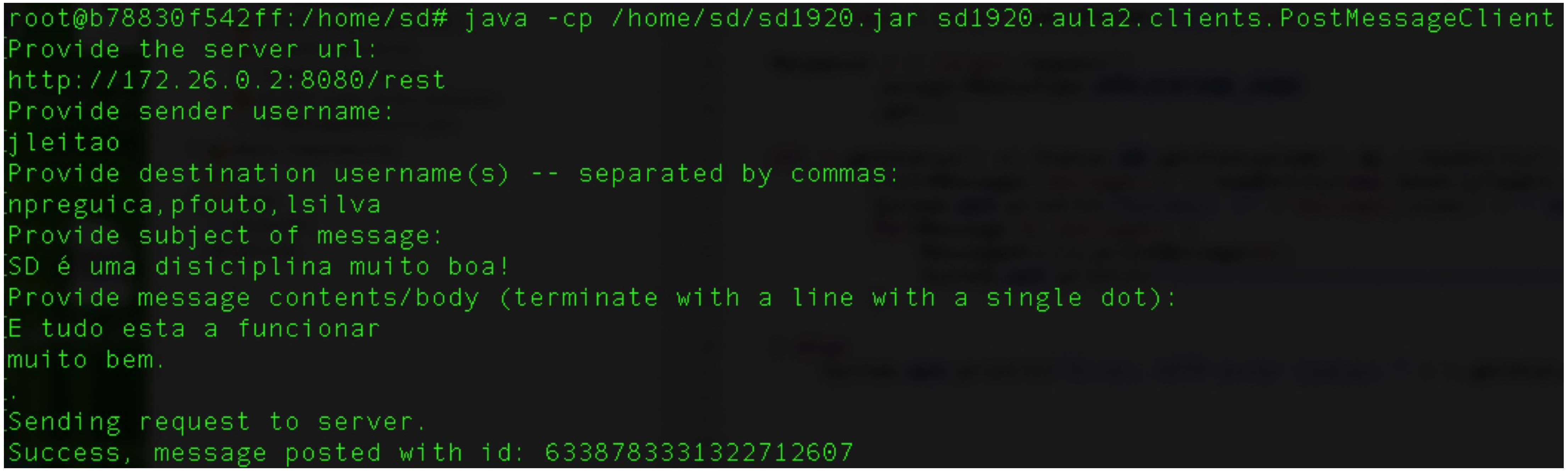
4. **Run another container in interactive mode (to execute clients) <u>in a second terminal window</u>**
   - docker run -it --network sdnet sd1920-aula2-xxxxx-yyyyy /bin/bash



```
➜ docker run -it --network sdnet sd1920-aula2-xxxxx-yyyyy /bin/bash

root@b78830f542ff:/home/sd#
```

Testing with Docker

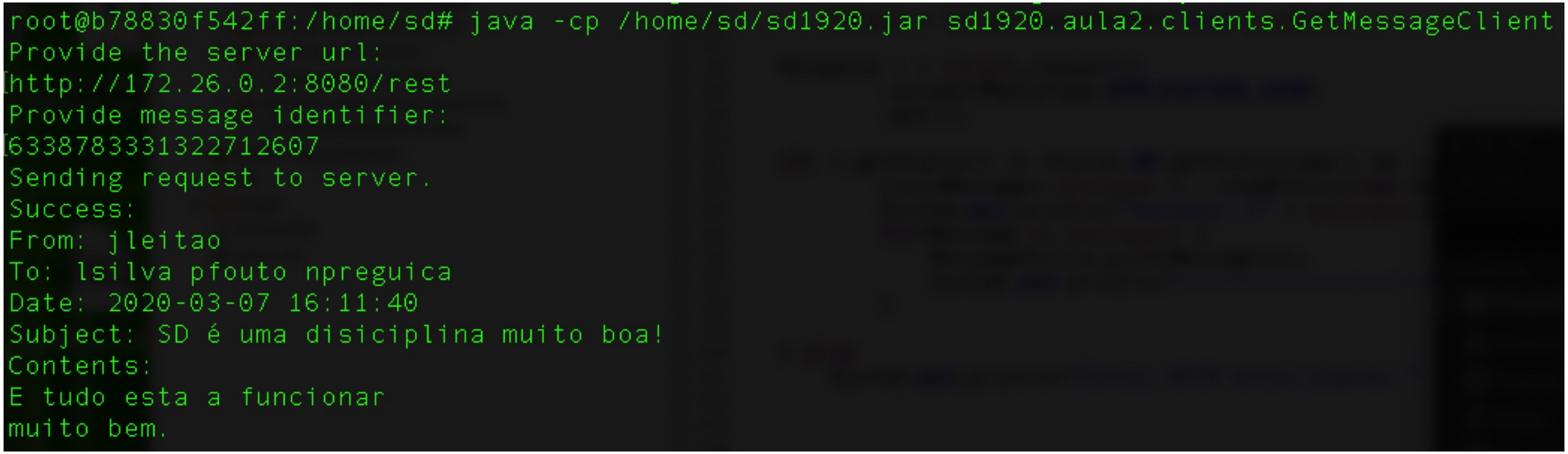
5. **Run the client to post a message in the second container**
   - java -cp /home/sd/sd1920.jar sd1920.aula2.clients.PostMessageClient



```
root@b78830f542ff:/home/sd# java -cp /home/sd/sd1920.jar sd1920.aula2.clients.PostMessageClient
Provide the server url:
http://172.26.0.2:8080/rest
Provide sender username:
jleitao
Provide destination username(s) -- separated by commas:
npreguica,pfouto,lsilva
Provide subject of message:
SD é uma disiciplina muito boa!
Provide message contents/body (terminate with a line with a single dot):
E tudo esta a funcionar
muito bem.
.
Sending request to server.
Success, message posted with id: 6338783331322712607
```

TESTING WITH DOCKER

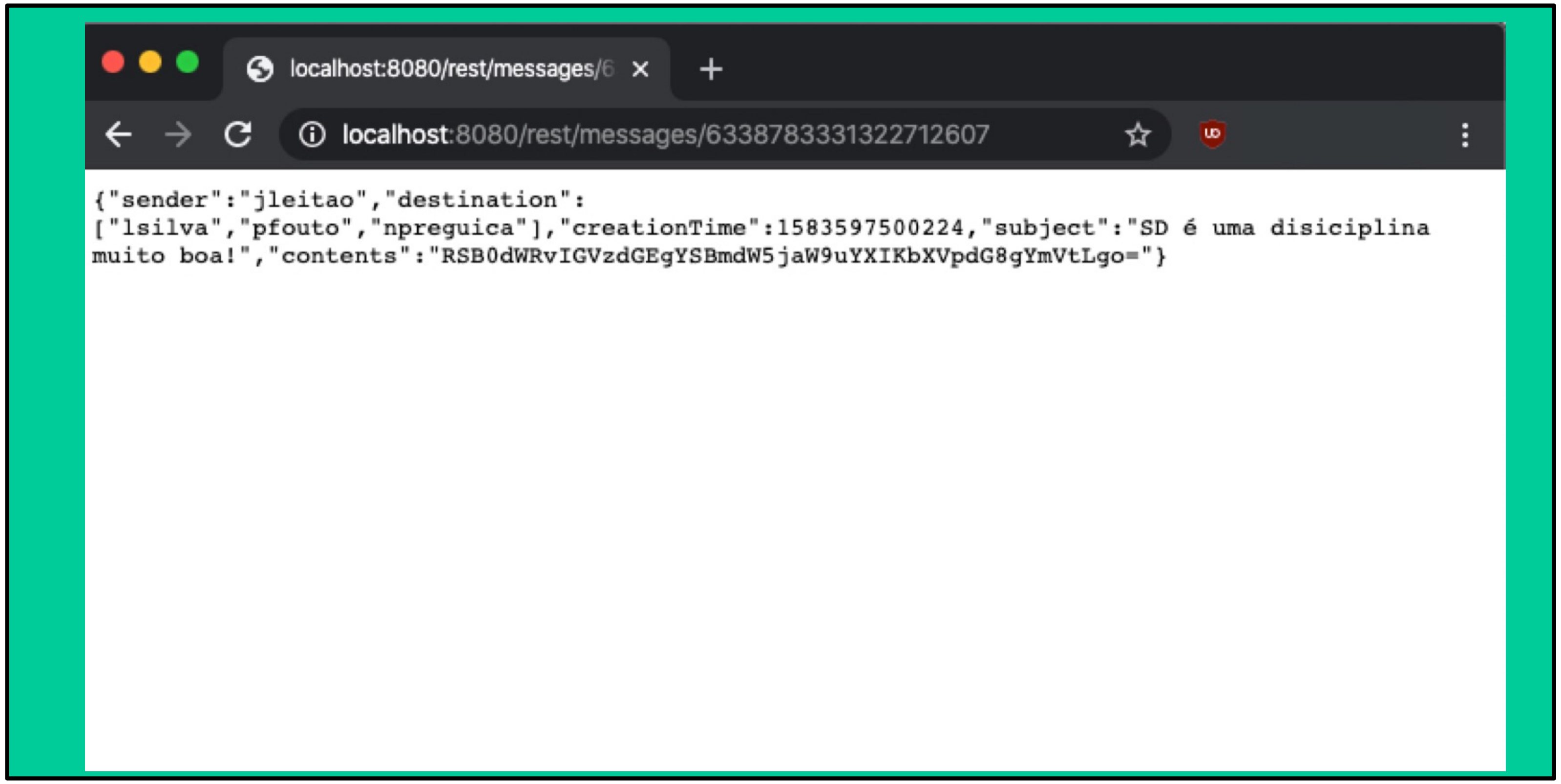
6. **Run the client to get a message <u>in the second container</u>**
   - java -cp /home/sd/sd1920.jar sd1920.aula2.clients.GetMessageClient



```
root@b78830f542ff:/home/sd# java -cp /home/sd/sd1920.jar sd1920.aula2.clients.GetMessageClient
Provide the server url:
http://172.26.0.2:8080/rest
Provide message identifier:
6338783331322712607
Sending request to server.
Success:
From: jleitao
To: lsilva pfouto npreguica
Date: 2020-03-07 16:11:40
Subject: SD é uma disiciplina muito boa!
Contents:
E tudo esta a funcionar
muito bem.
```

TESTING WITH DOCKER

7. **Use your browser to access your service (optional)**


localhost:8080/rest/messages/6
localhost:8080/rest/messages/6338783331322712607

```
{"sender":"jleitao","destination":
["lsilva","pfouto","npreguica"],"creationTime":1583597500224,"subject":"SD é uma disiciplina
muito boa!","contents":"RSB0dWRvIGVzdGEgYSBmdW5jaW9uYXIKbXVpdG8gYmVtLgo="}
```

8. **Try the other clients that are provided in the second container (optional)**

EXERCISE

1. Complete the two operations missing in the server (*deleteMessage* is a DELETE, and *removeFromUserInbox* is a PUT). Don't forget to:
   1. Add the Jersey annotations where you need them.
   2. Complete the implementation of the service
   3. You cannot delete something that does not exists.
2. Create two new clients, based on the provided ones, to exercise these two operations above.
3. Test your implementation using docker.
4. Integrate the Discovery class from last week to enable all clients to obtain the server URL automatically.