DISTRIBUTED SYSTEMS

Lab 8

João Leitão, Sérgio Duarte, Pedro Camponês

GOALS

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

- Understand what is HTTPS and SSL/TLS
- Know how to generate a keystore with server cryptographic keys
- Know how to generate a truststore with root certificates and the certificate for your server
- Know how to develop a REST server using https in Java
- Know how to develop a gRPC server using TLS in Java
- Know how to modify your REST clients to use https
- Know how to modify your gRPC clients to use TLS

GOALS

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

- **Understand what is HTTPS and SSL/TLS**
- Know how to generate a keystore with server cryptographic keys
- Know how to generate a truststore with root certificates and the certificate for your server
- Know how to develop a REST server using https in Java
- Know how to develop a gRPC server using TLS in Java
- Know how to modify your REST clients to use https
- Know how to modify your gRPC clients to use TLS

SSL/TLS

TLS is a cryptographic protocol suite to ensure secure communications over insecure networks. It is the second version of the SSL protocol (that was discovered to have vulnerabilities some years ago).

- TLS in addition to secure the communication (by avoiding sending messages in clear text) also allows to authenticate the identity of the server.
- Optionally, it can also authenticate the identity of the client (this is a feature rarely used that we will not explore in this course)
- TLS is the basis for the HTTPS protocol that allows secure accesses to content and services in the web.

SSL/TLS

TLS is a cryptographic protocol suite to ensure secure communications over insecure networks. It is the second version of the SSL protocol (that was discovered to have vulnerabilities some years ago).

- T
  s
  t
- C
  (
  c
- T
  a

In summary, TLS provides the following security guarantees:

Privacy - data streams are encrypted;
Integrity - data streams are protected against tampering;
Authentication - identity of servers and clients (optionally) is asserted.

HTTPS

HTTPS is the name of the protocol that results from the exchange of HTTP messages on top of TLS (secure) connections.

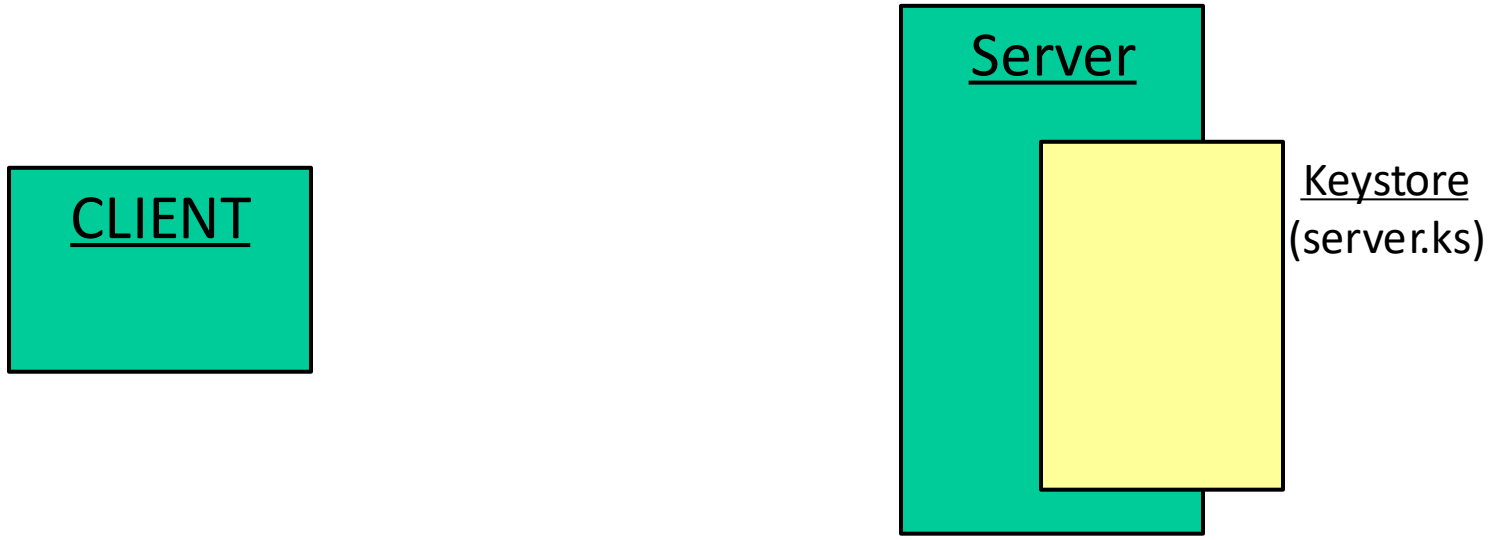
It is now a standard for accessing content in the web:

- It verifies the identity of the server, mitigating attacks such as Phishing and Man-in-the-Middle
- It ensures the privacy and integrity of all data exchanged between clients and servers (both requests and replies)

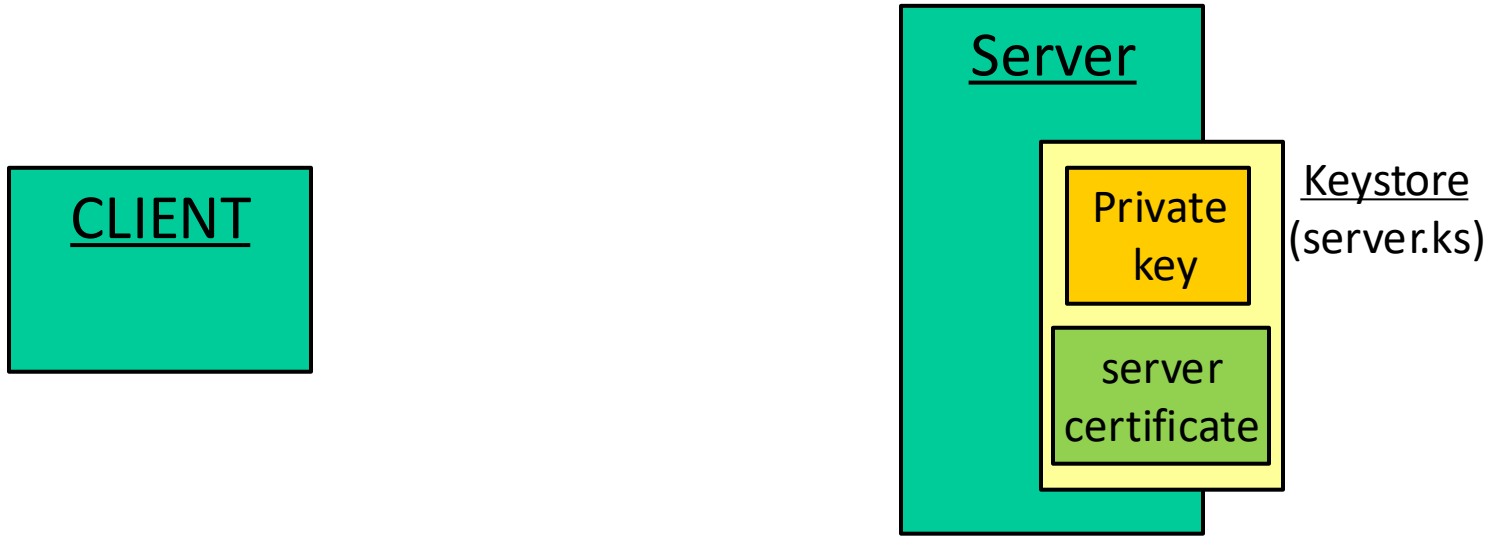
Important for instance, to enable sending passwords in URLs (as these will not be observable by someone inspecting the traffic departing the client machine)

HTTPS (HIGH LEVEL OVERVIEW)

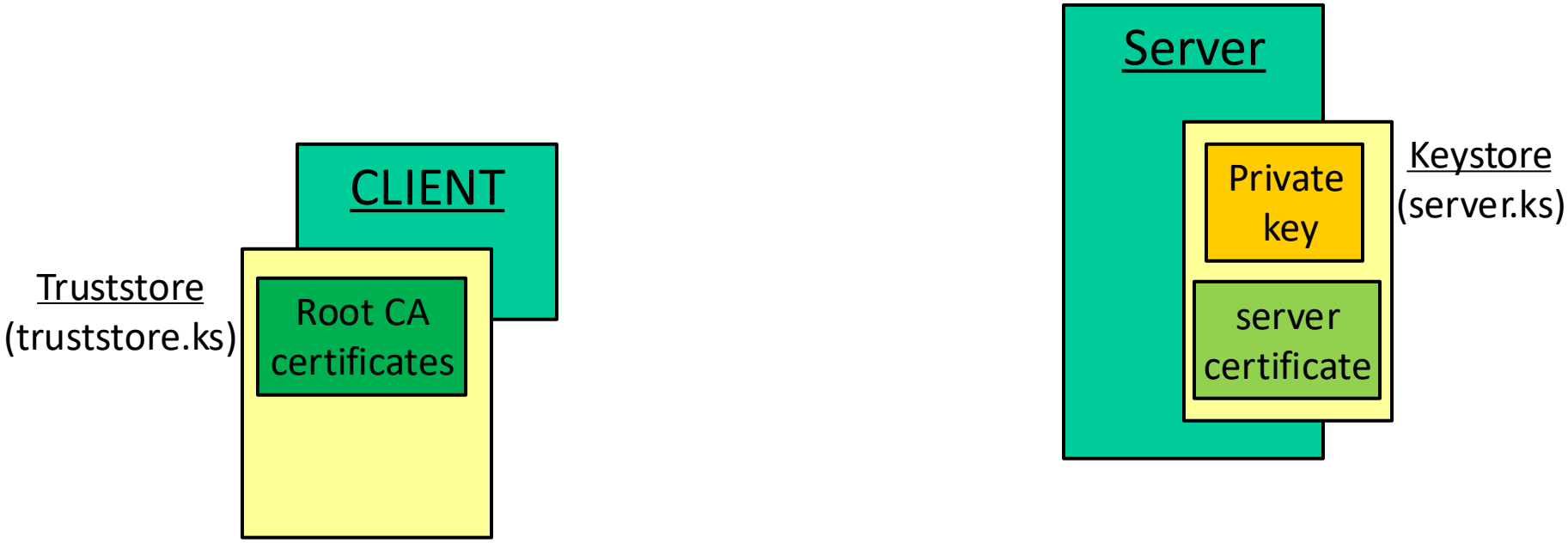
HTTPS (HIGH LEVEL OVERVIEW)



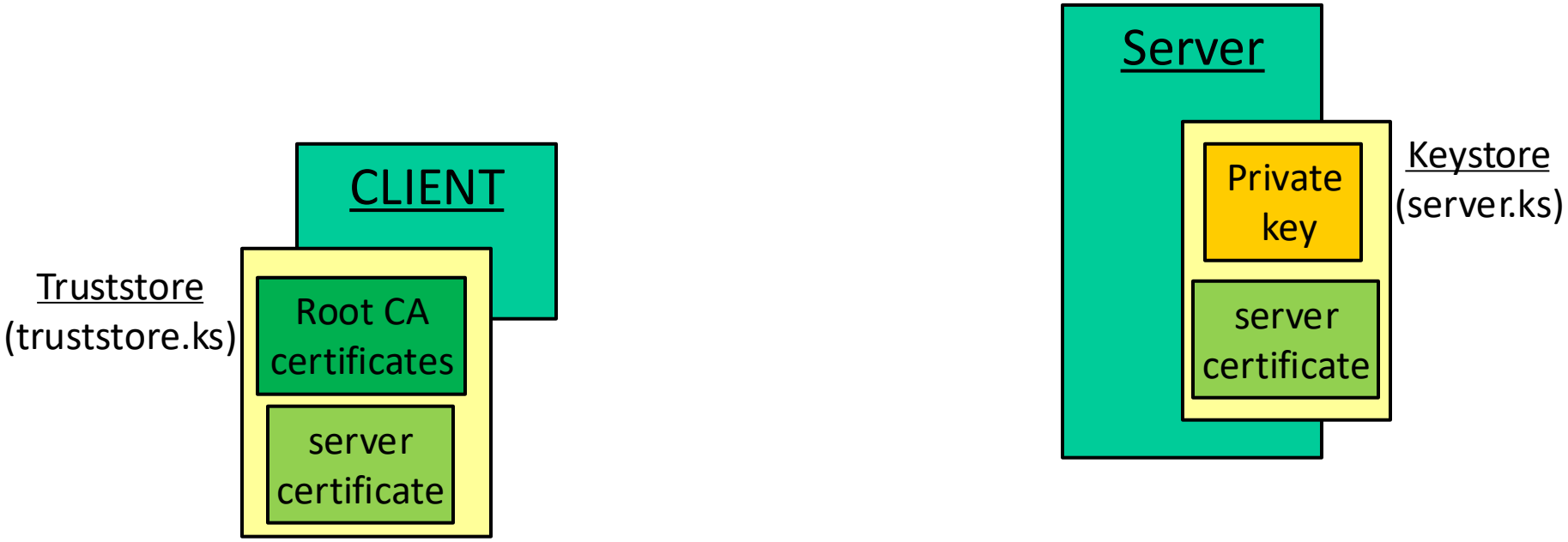
HTTPS (HIGH LEVEL OVERVIEW)



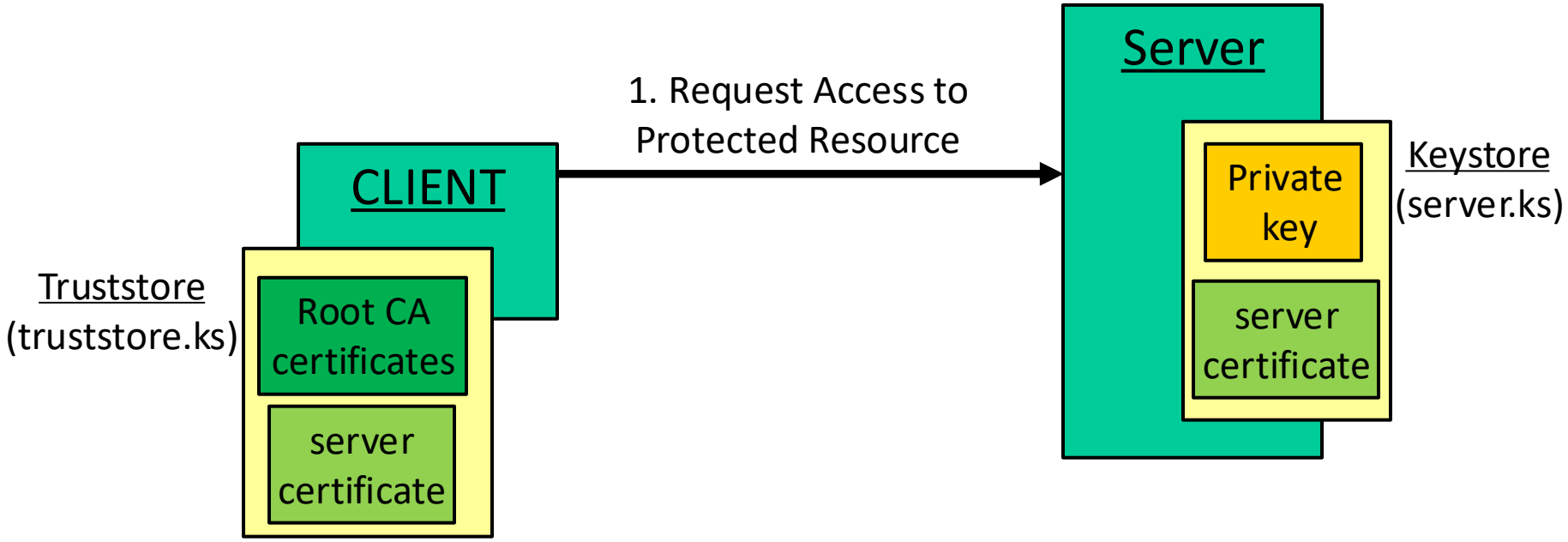
HTTPS (HIGH LEVEL OVERVIEW)



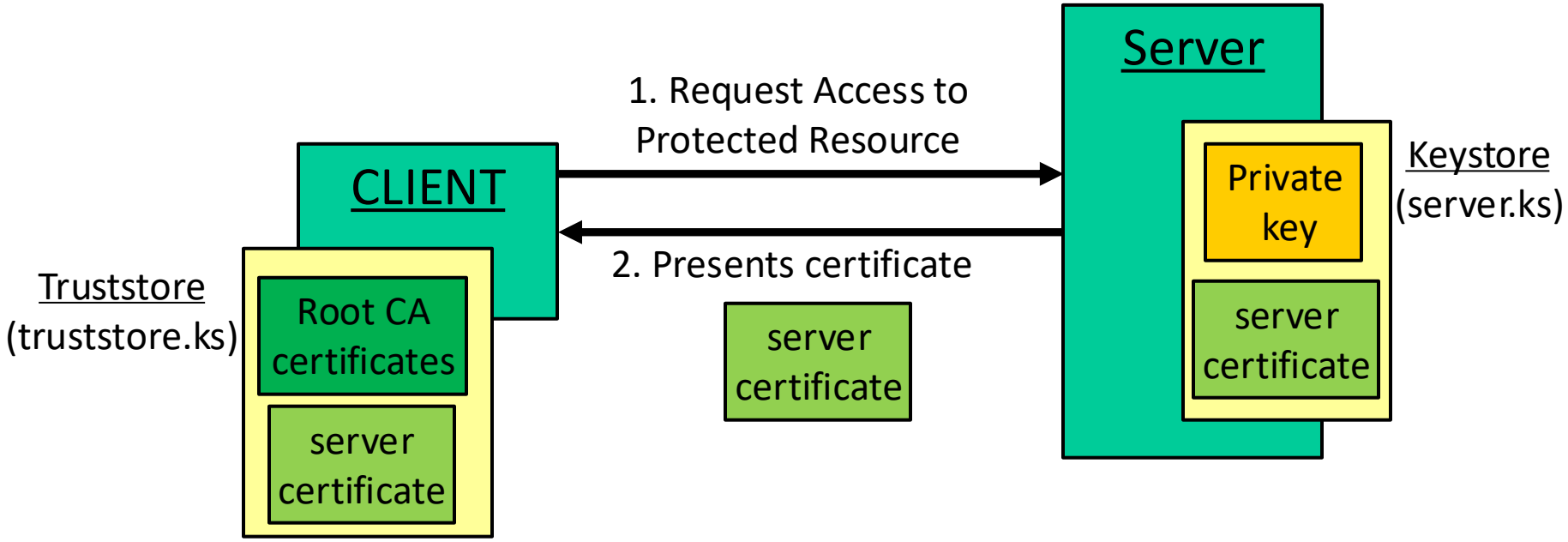
HTTPS (HIGH LEVEL OVERVIEW)



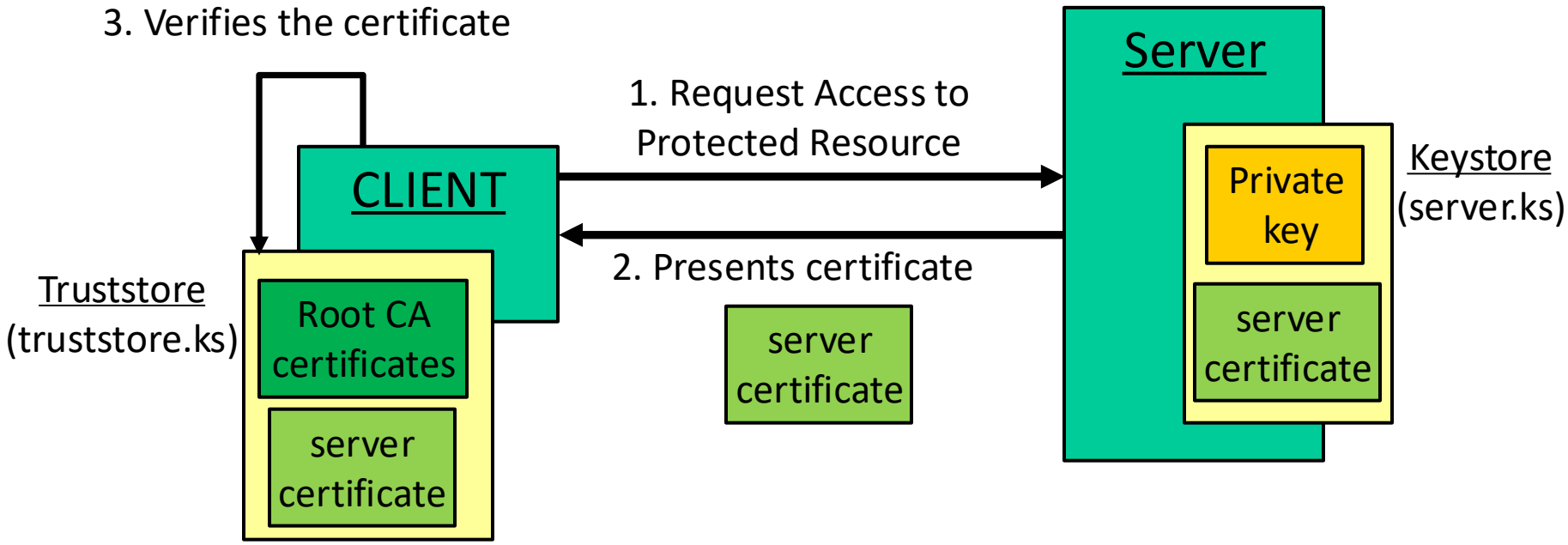
HTTPS (HIGH LEVEL OVERVIEW)



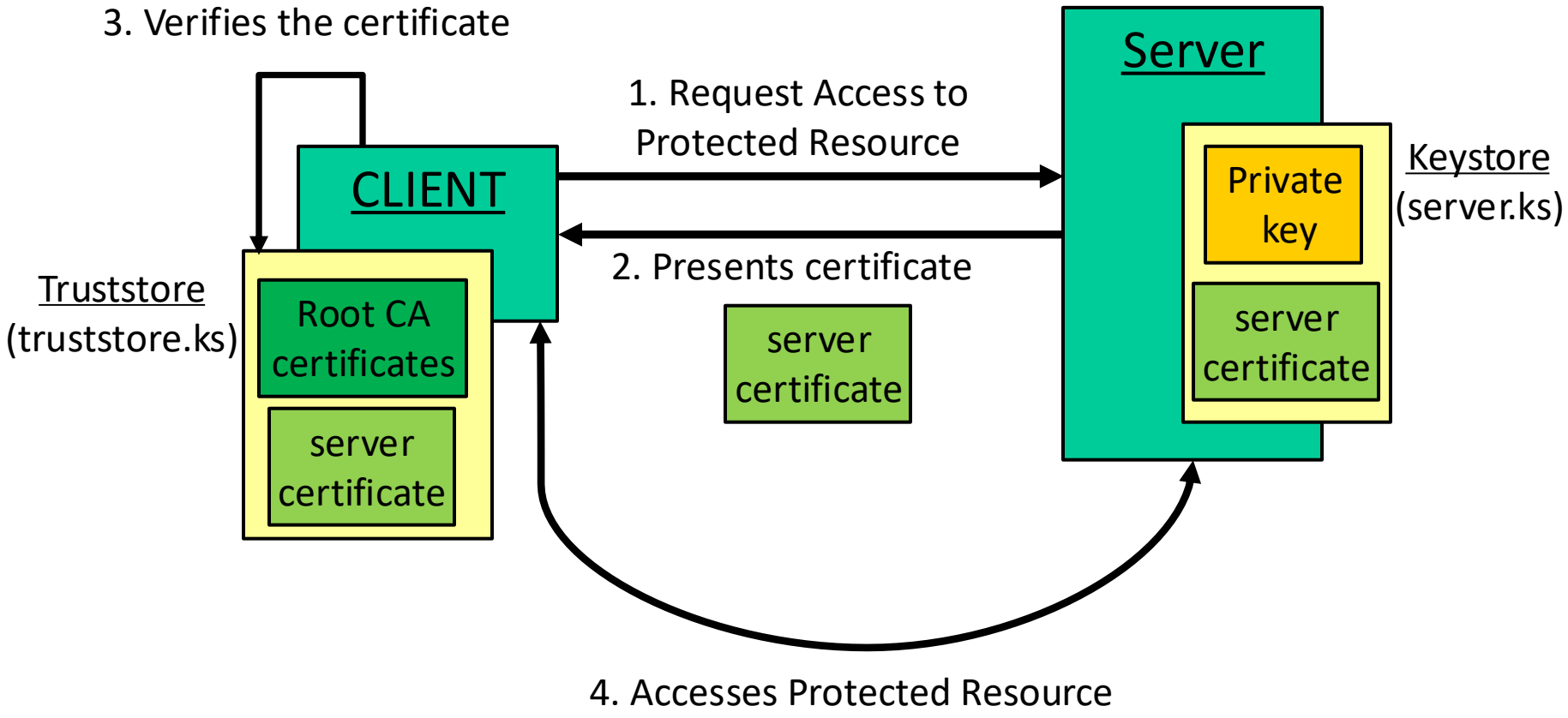
HTTPS (HIGH LEVEL OVERVIEW)



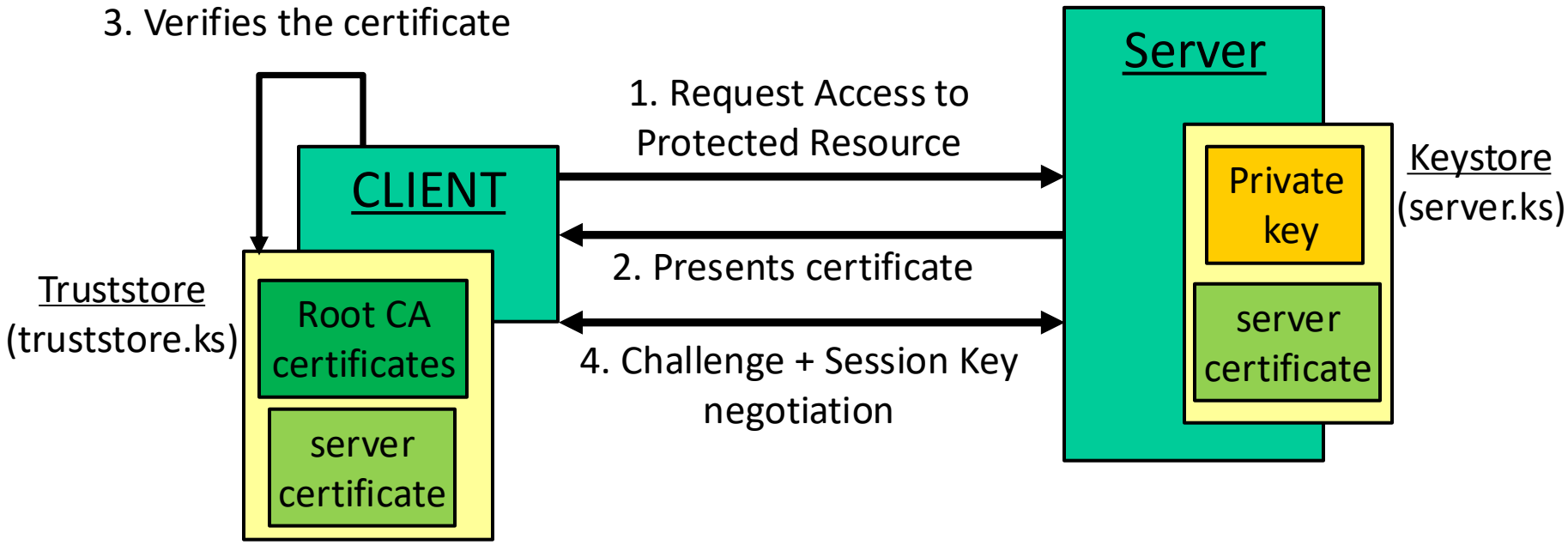
HTTPS (HIGH LEVEL OVERVIEW)



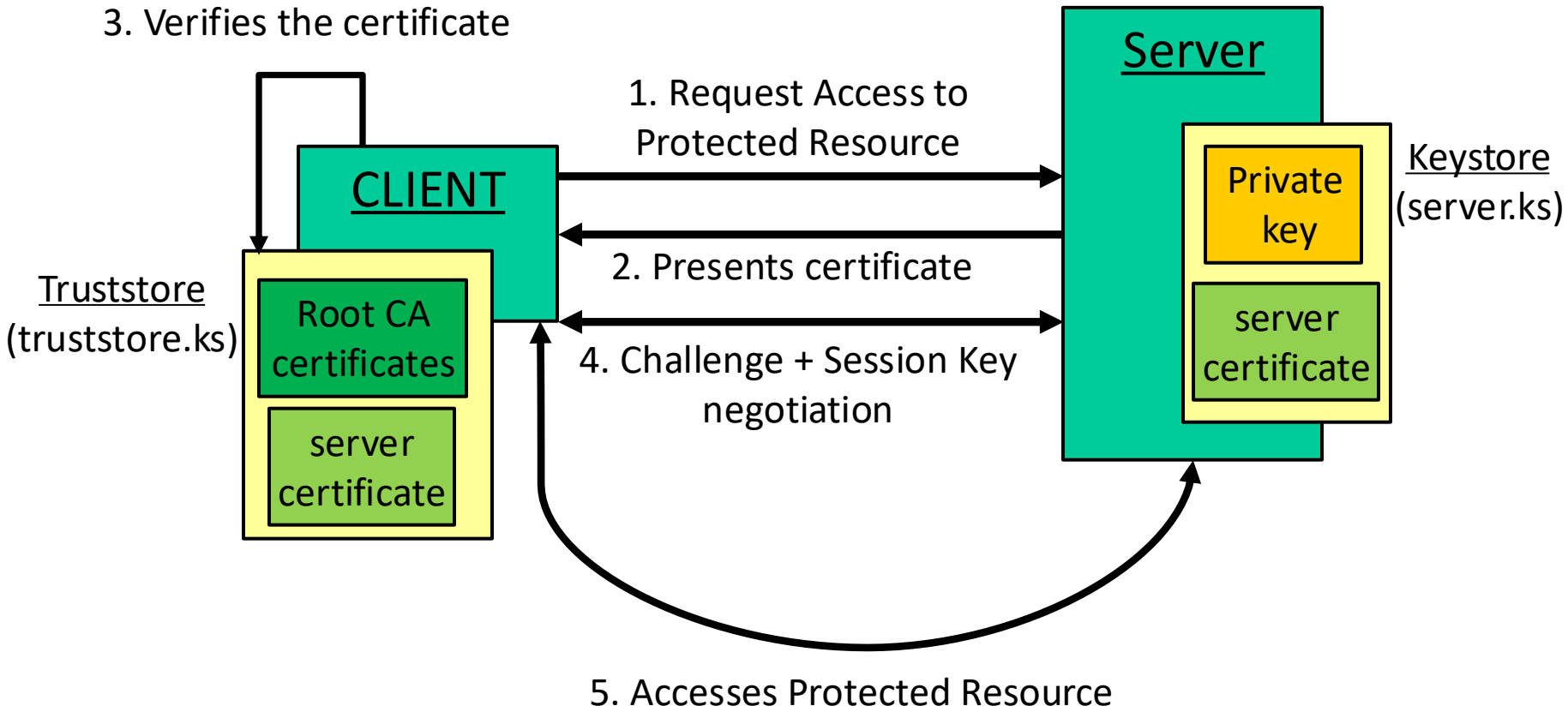
HTTPS (HIGH LEVEL OVERVIEW)



HTTPS (HIGH LEVEL OVERVIEW)

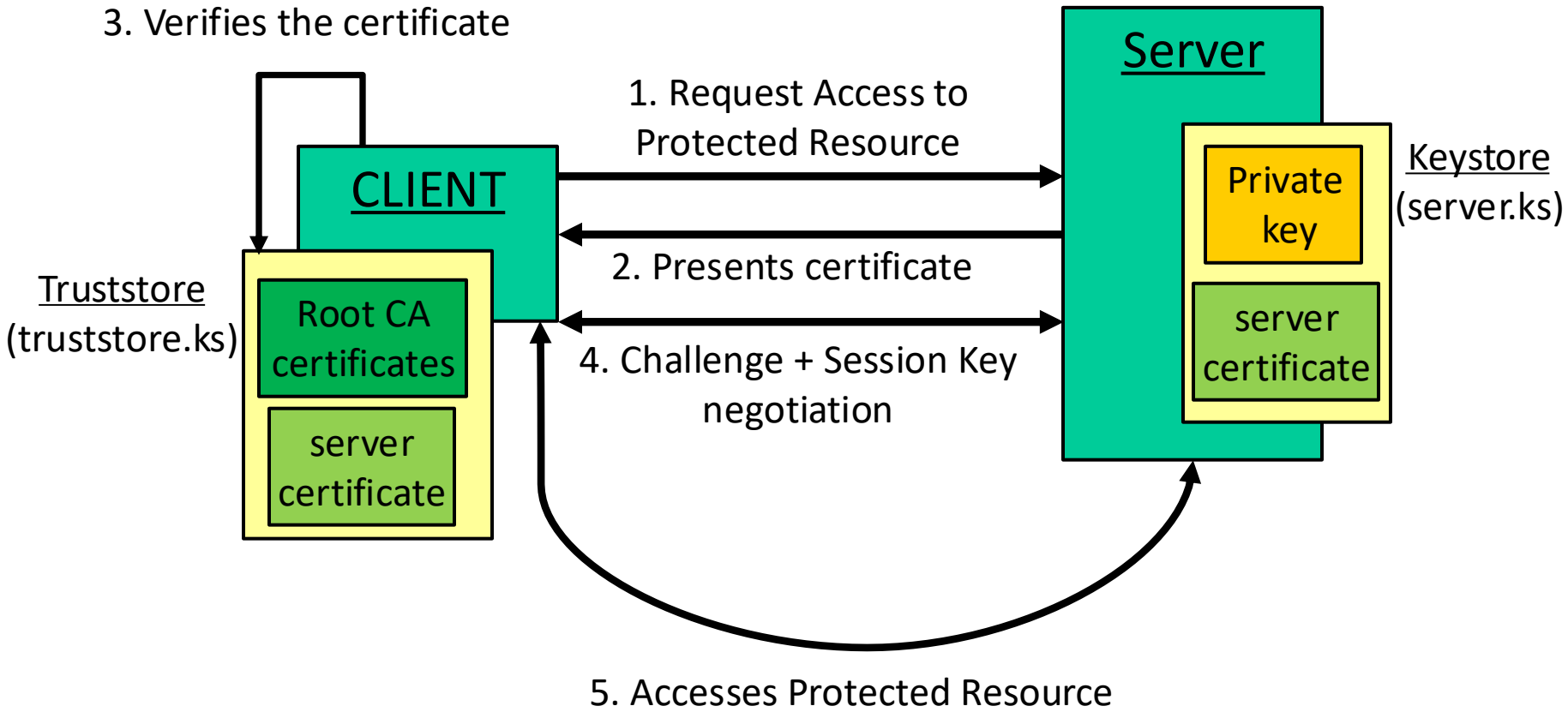


HTTPS (HIGH LEVEL OVERVIEW)



HTTPS (HIGH LEVEL OVERVIEW)

Naturally, the client might be optionally obligated to authenticate itself in between steps 3 and 4 (e.g., by presenting its own certificate using TLS, or by providing username/password at the application level)



GOALS

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

- Understand what is HTTPS and SSL/TLS
- **Know how to generate a keystore with server cryptographic keys**
- **Know how to generate a truststore with root certificates and the certificate for your server**
- Know how to develop a REST server using https in Java
- Know how to develop a gRPC server using TLS in Java
- Know how to modify your REST clients to use https
- Know how to modify your gRPC clients to use TLS

GENERATING KEYSTORES AND TRUSTSTORES

From the previous explanation, you should have figured out that we need to create the keystore and truststore used by the servers and clients respectively.

- The authentication of the server in SSL/TLS uses public key certificates (https://en.wikipedia.org/wiki/Public_key_certificate)
- In the context of Java, both certificates and (private) keys are stored in a keystore.
- The keytool is the command in the java environment that allows to manipulate keystores, certificates and keys.

GENERATING THE SERVER KEYSTORE

CLIENT

Server

To generate the server keystore (which will include the server private key and certificate you should use the following command (in the root directory of your project):

```
keytool -ext SAN=dns:<server-name> -genkey -alias <server-name> -keyalg
RSA -validity 365 -keystore <keystore-filename> -storetype pkcs12
```

GENERATING THE SERVER KEYSTORE

CLIENT

Server

To generate the server keystore (which will include the server private key and certificate you should use the following command (in the root directory of your project):

```
keytool -ext SAN=dns:users -genkey -alias users -keyalg RSA -validity 365
-keystore users-server.ks -storetype pkcs12
```

For instance, assuming that the server is named **users** and that it will run on a machine/container named **users**

GENERATING THE SERVER KEYSTORE

CLIENT

Server

To generate the server keystore (which will include the server private key and certificate you should use the following command (in the root directory of your project):

```
keytool -ext SAN=dns:users -genkey -alias users -keyalg RSA -validity 365
-keystore users-server.ks -storetype pkcs12
```

This command generates a public key certificate identified by name 'server', that is valid for 365 days, that will be stored in a keystore in a file named 'server.ks', being signed by its own key (self-signed certificate)

GENERATING THE SERVER KEYSTORE

CLIENT

Server

```
jleitao@10-139-25-153 lab8 % keytool -ext SAN=dns:users -genkey -alias users -keyalg RSA -validity 365 -keystore users-server.ks -s
toretype pkcs12

Enter keystore password:
Re-enter new password:
Enter the distinguished name. Provide a single dot (.) to leave a sub-component empty or press ENTER to use the default value in br
aces.
What is your first and last name?
  [Unknown]:  users
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  yes

Generating 3,072 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 365 days
        for: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
jleitao@10-139-25-153 lab8 %
```

GENERATING THE SERVER KEYSTORE

CLIENT

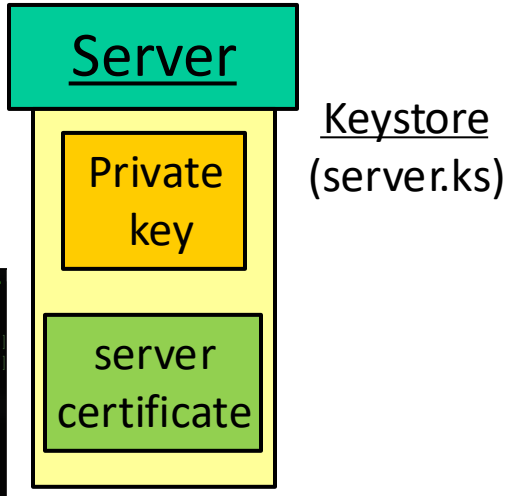
Server

```
jleitao@10-139-25-153 lab8 % keytool -ext SAN=dns:users -genkey -alias users -keyalg RSA -validity 365 -keystore users-server.ks -s
toretype pkcs12

Enter keystore password:
Re-enter new password:
Enter the distinguished name. Provide a single dot (.)                                                    lue in br
aces.
What is your first and last name?
  [Unknown]:  users
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  yes

Generating 3,072 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 365 days
        for: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
jleitao@10-139-25-153 lab8 %
```

It will ask you for a password to protect the keystore. I have used: *password*

GENERATING THE SERVER KEYSTORE

CLIENT




```
jleitao@10-139-25-153 lab8 % keytool -ext SAN=dns:users -genkey -alias users -keyalg RSA -validity 365 -keystore users-server.ks -s
toretype pkcs12

Enter keystore password:
Re-enter new password:
Enter the distinguished name. Provide a single dot (.) to leave a sub-component empty or press ENTER to use the default value in br
aces.
What is your first and last name?
  [Unknown]:  users
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  yes

Generating 3,072 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 365 days
        for: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
jleitao@10-139-25-153 lab8 %
```

After executing this command successfully, you will have created the server keystore (in a file named: users-*server.ks*) that includes within it the server private key and the server public key certificate.

GENERATING THE CLIENT TRUSTSTORE

We now need to generate the client truststore. Notice that a truststore is similar to a keystore with the exception that it only stores public key certificates belonging to other entities.

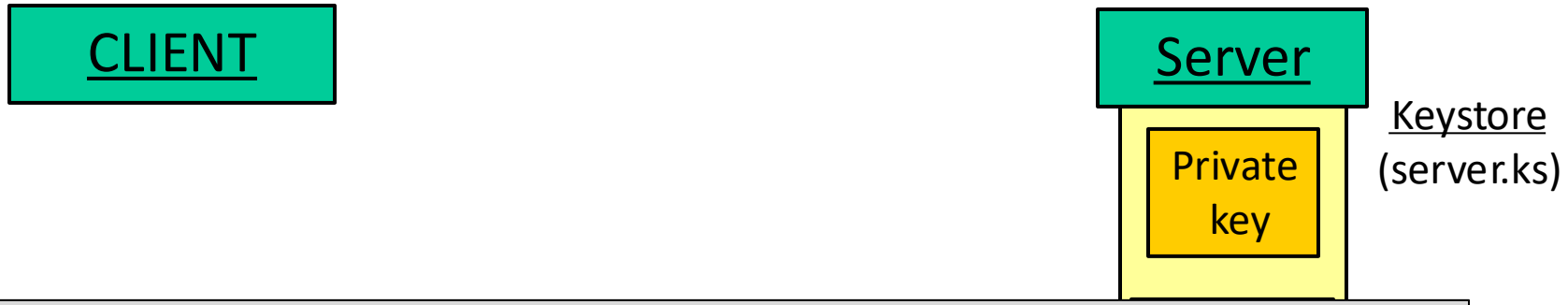
Since we want to allow our clients to interact with any server in the world, its truststore will have to contain the certificates of all Root Certification Authorities (such as VerySign or LetsEncrypt).

GENERATING THE CLIENT TRUSTSTORE

To do that we will initialize the client truststore with a copy of the Java environment **cacerts**, the default truststore that contains all Root CA certificates with the command:

```
cp /Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/lib/security/cacerts truststore.ks
```

GENERATING THE CLIENT TRUSTSTORE



CACERTS

The default Java truststore cacerts contains just a list of root CA certificates.

These are certificates issued by "Certification Authorities" that Java trusts implicitly.

They are stored in a file named cacerts included in every JDK and JRE distribution.

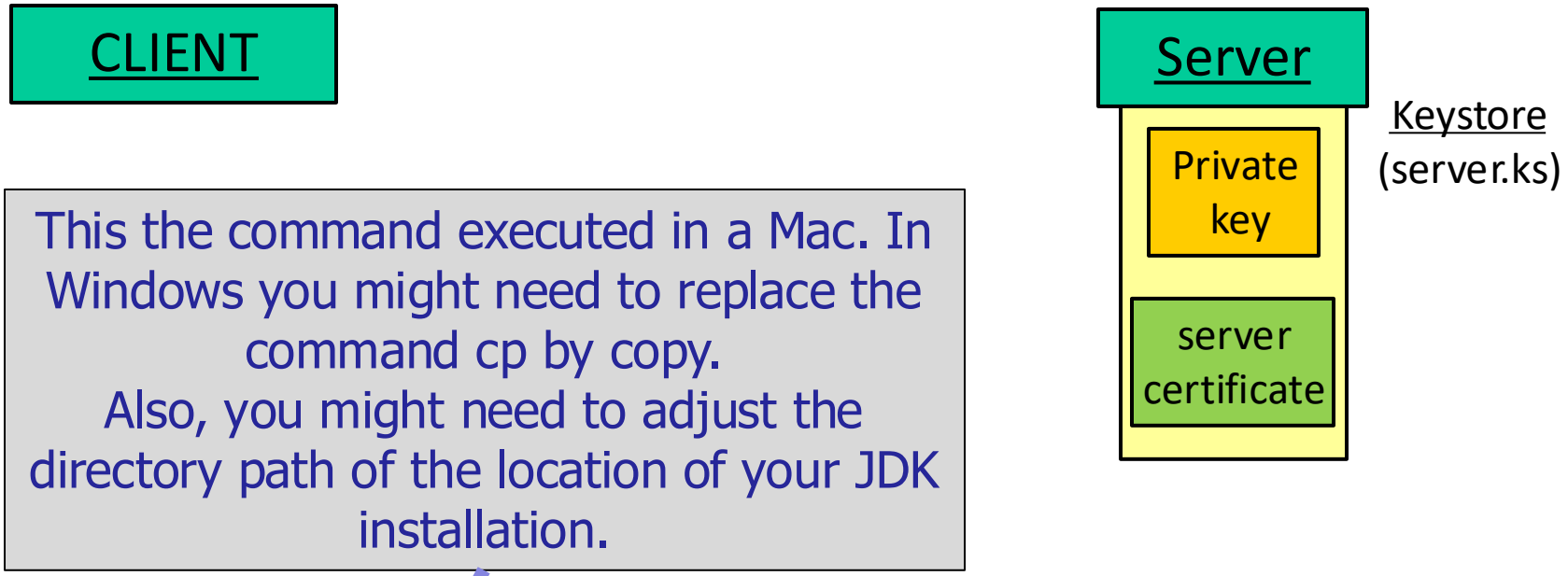
The default cacertspassword is changeit.

To do
the Ja
contai

```
cp /Li
17.jdk
```

opy of
hat
e.ks

GENERATING THE CLIENT TRUSTSTORE



To do that we will initialize the client truststore with a copy of the Java environment cacerts, the default truststore that contains all Root CA certificates with the command:

```
cp /Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/lib/security/cacerts truststore.ks
```

GENERATING THE CLIENT TRUSTSTORE

After executing this command successfully, you will have created the truststore for the client (in a file named: truststore.ks) containing all Root CA certificates. Since this a copy of the Java cacerts, it is protected by the password:

changeit

```
cp /Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/lib/security/cacerts truststore.ks
```

EXPORTING THE SERVER CERTIFICATE

We still need to insert the server certificate in the client truststore.

However, to do that we will need first to export the server certificate from the server *keystore* to a temporary file.

EXPORTING THE SERVER CERTIFICATE

To export the server certificate, we use again keytool. You should run the following command:

```
keytool -exportcert -alias <server-name> -keystore
<keystore-file> -file <certificate-file>
```

EXPORTING THE SERVER CERTIFICATE

To export the server certificate, we use again keytool. You should run the following command:

```
keytool -exportcert -alias users -keystore users-server.ks -file users.cert
```

This command exports the certificate of entity named 'users' from the keystore file 'users-server.ks' into a temporary file named 'users.cert'

EXPORTING THE SERVER CERTIFICATE

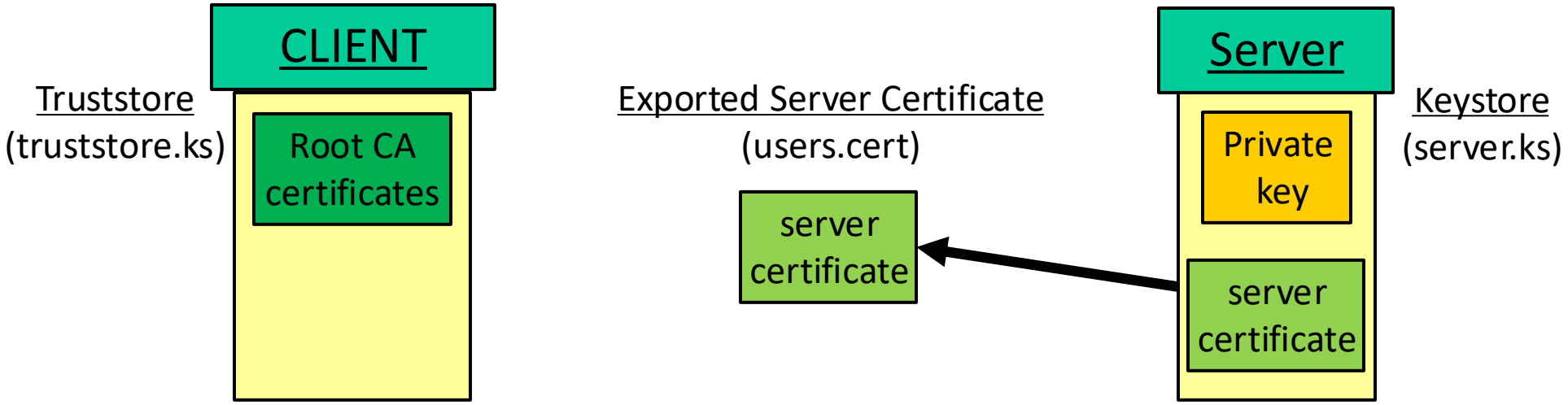

```
jleitao@10-139-25-153 lab8 % keytool -exportcert -alias users -keystore users-server.ks -file users.cert

Enter keystore password:
Certificate stored in file <users.cert>
```

EXPORTING THE SERVER CERTIFICATE

It will ask you for the password that protects the keystore file.

```
jleitao@10-139-25-153 lab8 % keytool -exportcert -alias users -keystore users-server.ks -file users.cert

Enter keystore password:
Certificate stored in file <users.cert>
```

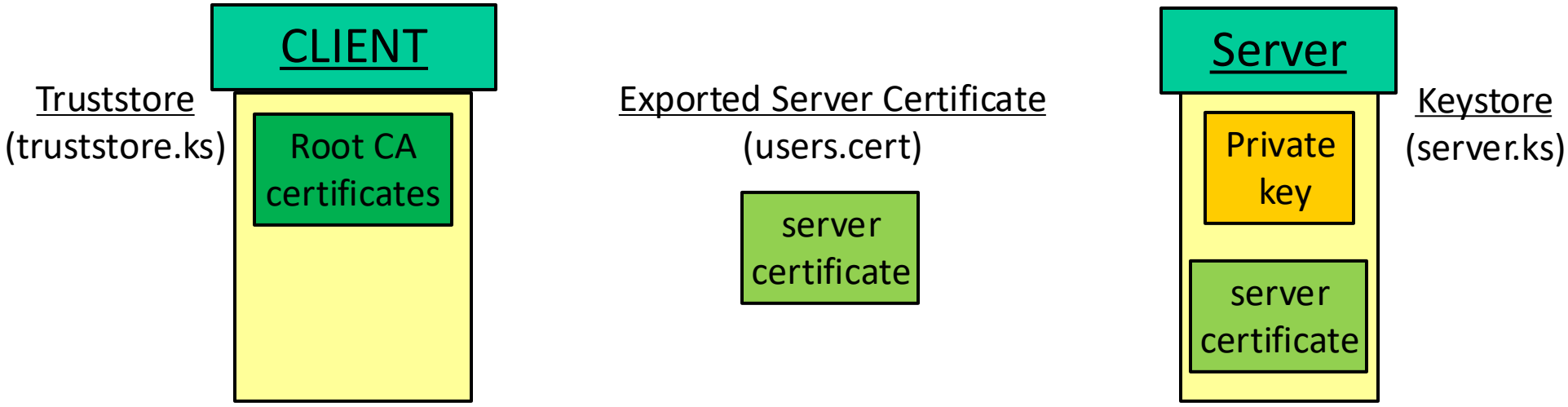
EXPORTING THE SERVER CERTIFICATE



After executing this command successfully, you will have created a temporary file (users.cert) with the server certificate

```
jleitao@10-139-25-153 lab8 % keytool -exportcert -alias users -keystore users-server.ks -file users.cert

Enter keystore password:
Certificate stored in file <users.cert>
```

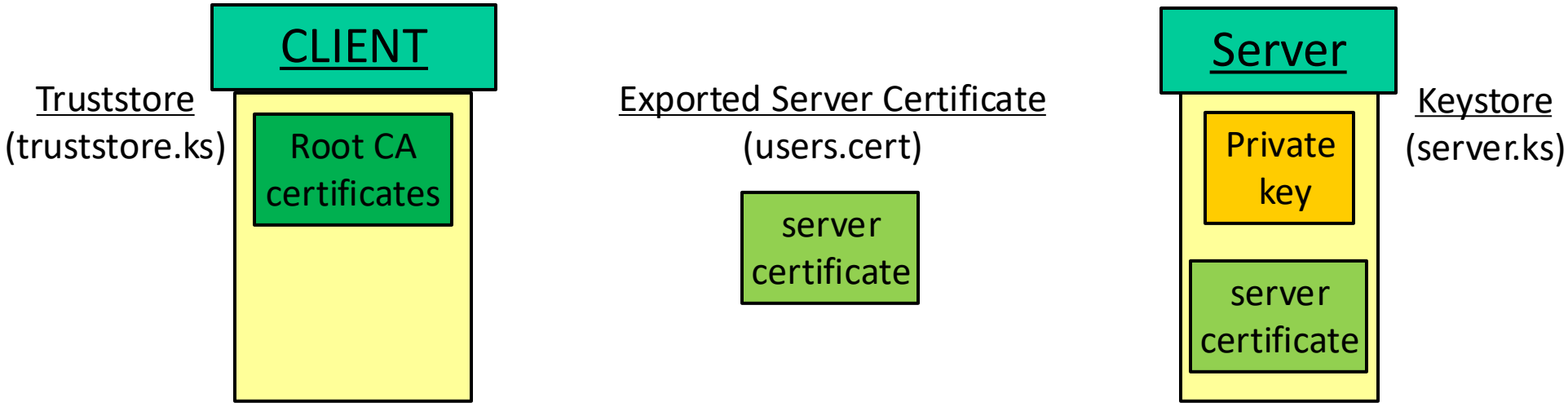
IMPORTING THE SERVER CERTIFICATE



Now that we have exported the server certificate, we just need to add it to the client truststore. Again, we will use keytool to do this:

```
keytool -importcert -file <certificate-file> -alias
<server-name> -keystore <keystore-file>
```

IMPORTING THE SERVER CERTIFICATE



Now that we have exported the server certificate, we just need to add it to the client truststore. Again, we will use keytool to do this:

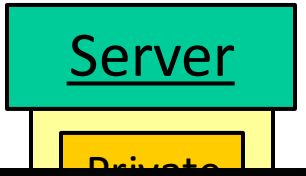
```
keytool -importcert -file users.cert -alias users -keystore truststore.ks
```

This command imports the certificate of entity named 'users', located in file 'users.cert', into the truststore (i.e., a keystore) named 'truststore.ks'

IMPORTING THE SERVER CERTIFICATE

Truststore

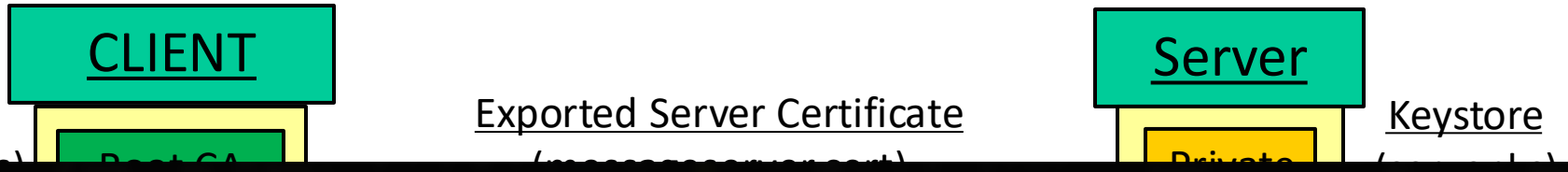
Exported Server Certificate



Keystore

```
jleitao@10-139-25-153 lab8 % keytool -importcert -file users.cert -alias users -keystore truststore.ks

Enter keystore password:
Owner: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 3c1afdc8f39a811b
Valid from: Sun Apr 27 15:48:08 WEST 2025 until: Mon Apr 27 15:48:08 WEST 2026
Certificate fingerprints:
         SHA1: E7:25:93:15:51:B0:5A:66:A2:7B:BB:A3:12:F1:DE:A3:EA:D1:04:31
         SHA256: 03:4D:1E:22:FF:29:F0:09:73:A1:F9:98:74:24:01:40:D8:B3:A0:AB:FE:58:2F:03:F5:83:CB:C3:55:EC:38:FB
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 3072-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=false
SubjectAlternativeName [
  DNSName: users
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 4B 61 0B 79 E4 6A 20 01   D4 9B 2F 12 E3 E2 F2 F1  Ka.y.j .../.....
0010: 2D 1D F8 34                                        -..4
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

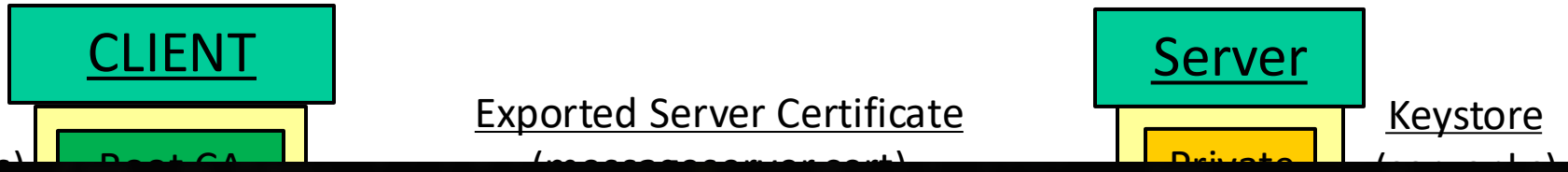
IMPORTING THE SERVER CERTIFICATE




```
jleitao@10-139-25-153 lab8 % keytool -importcert -file users.cert -alias users -keystore truststore.ks

Enter keystore password:
Owner: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 3c1afdc8f39a811b
Valid from: Sun Apr 27 15:48:08 WEST 2025 until: Mon Apr 27 15:48:08 WEST 2026
Certificate fingerprints:
         SHA1: E7:25:93:15:51:B0:5A:66:A2:7B:BB:A3:12:F1:DE:A3:EA:D1:04:31
         SHA256: 03:4D:1E:22:FF:29:F0:09:73:A1:F9:98:74:24:01:40:D8:B3:A0:AB:FE:58:2F:03:F5:83:CB:C3:55:EC:38:FB
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 3072-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=false
SubjectAlternativeName [
  DNSName: users
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 4B 61 0B 79 E4 6A 20 01   D4 9B 2F 12 E3 E2 F2 F1  Ka.y.j ..../.....
0010: 2D 1D F8 34                                        -..4
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

It will ask you for the password that protects the truststore file.
As said before this is the password of the original cacerts file: *changeit*

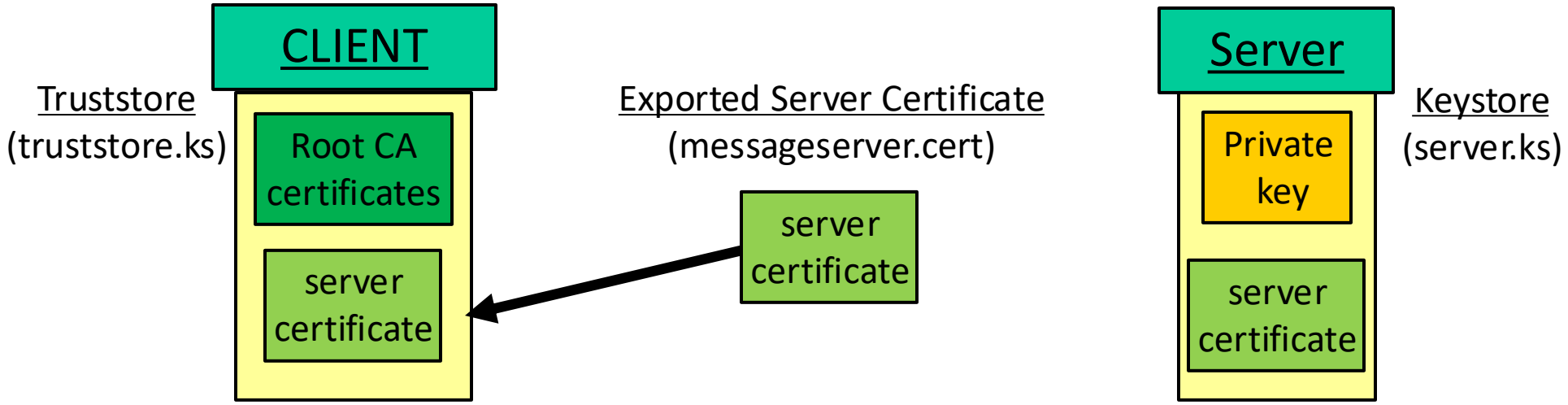
IMPORTING THE SERVER CERTIFICATE




```
jleitao@10-139-25-153 lab8 % keytool -importcert -file users.cert -alias users -keystore truststore.ks

Enter keystore password:
Owner: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 3c1afdc8f39a811b
Valid from: Sun Apr 27 15:48:08 WEST 2025 until: Mon Apr 27 15:48:08 WEST 2026
Certificate fingerprints:
         SHA1: E7:25:93:15:51:B0:5A:66:A2:7B:BB:A3:12:F1:DE:A3:EA:D1:04:31
         SHA256: 03:4D:1E:22:FF:29:F0:09:73:A1:F9:98:74:24:01:40:D8:B3:A0:AB:FE:58:2F:03:F5:83:CB:C3:55:EC:38:FB
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 3072-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=false
SubjectAlternativeName [
  DNSName: users
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 4B 61 0B 79 E4 6A 20 01   D4 9B 2F 12 E3 E2
0010: 2D 1D F8 34
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

After showing you the fingerprint of the certificate that is being imported as well as the identity of the owner of the certificate you will have to confirm that you want to trust this certificate.

IMPORTING THE SERVER CERTIFICATE




```
jleitao@10-139-25-153 lab8 % keytool -importcert -file users.cert -alias users -keystore truststore.ks

Enter keystore password:
Owner: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=users, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 3c1afdc8f39a811b
Valid from: Sun Apr 27 15:48:08 WEST 2025 until: Mon Apr 27 15:48:08 WEST 2026
Certificate fingerprints:
         SHA1: E7:25:93:15:51:B0:5A:66:A2:7B:BB:A3:12:F1:DE:A3:EA:D1:04:31
         SHA256: 03:4D:1E:22:FF:29:F0:09:73:A1:F9:98:74:24:01:40:D8:B3:A0:AB:FE:58:2F:03:F5:83:CB:C3:55:EC:38:FB
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 3072-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=false
SubjectAlternativeName [
  DNSName: users
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 4B 61 0B 79 E4 6A 20 01   D4 9B 2F 12 E3 E2 F2 F1  Ka.y.j ..../.....
0010: 2D 1D F8 34                                        -..4
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

After executing this command successfully, you will have added the server certificate to the client truststore.

GOALS

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

- Understand what is HTTPS and SSL/TLS
- Know how to generate a keystore with server cryptographic keys
- Know how to generate a truststore with root certificates and the certificate for your server
- **Know how to develop a REST server using https in Java**
- **Know how to develop a gRPC server using TLS in Java**
- Know how to modify your REST clients to use https
- Know how to modify your gRPC clients to use TLS

REST SERVER AND HTTPS

Class containing the main of the Rest Server.

```
package lab8.impl.server.rest;

import java.net.InetAddress;..

public class UsersServer {

    private static Logger Log = Logger.getLogger(UsersServer.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 = "UsersService";
    private static final String SERVER_URI_FMT = "https://%s:%s/rest";

    public static void main(String[] args) {
        try {

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

        String hostname = InetAddress.getLocalHost().getHostName();
        String serverURI = String.format(SERVER_URI_FMT, hostname, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config,
                SSLContext.getDefault());

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

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

REST SERVER AND HTTPS

Class containing the main of the Rest Server.

```
package lab8.impl.server.rest;

import java.net.InetAddress;..

public class UsersServer {

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

    static {
        System.setProperty("java.net.preferIPv4S
        System.setProperty("java.util.logging.Si
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersService";
    private static final String SERVER_URI_FMT = "https://%s:%s/rest";

    public static void main(String[] args) {
        try {

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

        String hostname = InetAddress.getLocalHost().getHostName();
        String serverURI = String.format(SERVER_URI_FMT, hostname, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config,
                SSLContext.getDefault());

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

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

You need to change the URL of the server to have https:// instead of http://

REST SERVER AND HTTPS

Class containing the main of the Rest Server.

```
package lab8.impl.server.rest;

import java.net.InetAddress;..

public class UsersServer {

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

    static {
        System.setProperty("java.net.preferIPv4S
        System.setProperty("java.util.logging.Si
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersS
    private static final String SERVER_URI_FMT =

    public static void main(String[] args) {
        try {

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

        String hostname = InetAddress.getLocalHost().getHostName();
        String serverURI = String.format(SERVER_URI_FMT, hostname, PORT);
                SSLContext.getDefault());

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

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

The URL that clients use to contact the server can no longer have the IP address of the server and instead should have the hostname (that should be the <server-name> use to generate the server private key and certificate.

REST SERVER AND HTTPS

Class containing the main of the Rest Server.

```
package lab8.impl.server.rest;

import java.net.InetAddress;..

public class UsersServer {

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

    static {
        System.setProperty("java.net.preferI
        System.setProperty("java.util.loggir
    }

    public static final int PORT = 8080;
    public static final String SERVICE = "UsersService";
    private static final String SERVER_URI_FMT = "https://%s:%s/rest";

    public static void main(String[] args) {
        try {

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

        String hostname = InetAddress.getLocalHost().getHostName();
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config,
                SSLContext.getDefault());

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

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

The way that you start the HTTP server must be modified to include the default SSLContext (that is responsible for managing keystore and truststore)

REST SERVER AND HTTPS

Class containing the main of the Rest Server.

```
package lab8.impl.server.rest;

import java.net.InetAddress;..

public class UsersServer {

    private static Logger Log = Logger.getLogger(UsersServer.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 = "UsersService";
    private static final String SERVER_URI_FMT = "https://%s:%s/rest";

    public static void main(String[] args) {
        try {

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

        String hostname = InetAddress.getLocalHost().getHostName();
        String serverURI = String.format(SERVER_URI_FMT, hostname, PORT);
        JdkHttpServerFactory.createHttpServer( URI.create(serverURI), config,
                SSLContext.getDefault());

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

        //More code can be executed here...
        } catch( Exception e) {
            Log.severe(e.getMessage());
        }
    }
}
```

And these covers all changes to the server side in REST.

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;..

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

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

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

        String keyStoreFilename = System.getProperty("javax.net.ssl.keyStore");
        String keyStorePassword = System.getProperty("javax.net.ssl.keyStorePassword");

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFactory)
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostName(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grp

    private static Logger Log = Logger.getLogger(Users

    public static void main(String[] args) throws Exce

        String keyStoreFilename = System.getProperty('
        String keyStorePassword = System.getProperty('

        KeyStore keystore = KeyStore.getInstance(KeySt

        try(FileInputStream input = new FileInputStrea
            keystore.load(input, keyStorePassword.toCh
        }

        KeyManagerFactory keyManagerFactory = KeyManag
                KeyManagerFactory.getDefaultAlgorithm(
        keyManagerFactory.init(keystore, keyStorePassw

        GrpcUsersServerStub stub = new GrpcUsersServer

        SslContext context = GrpcSslContexts.configure
                SslContextBuilder.forServer(keyManager
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostName(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

By default, gRPC will use keys and certificates in PEM format (https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail).

It is possible to convert pkcs12 keys and certificates to PEM using the openssl (https://www.openssl.org/) command line tool.

Instead, we are going to take advantage of the integration of gRPC with the netty (https://netty.io/) communication library () that allows to use pkc12 keys.

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

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

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

        String keyStoreFilename = System.getProperty("javax.net.ssl.keyStore");
        String keyStorePassword = System.getProperty("javax.net.ssl.keyStorePassword");

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManage
                KeyManagerFactory.getDefaultAlgorithm()
        keyManagerFactory.init(keystore, keyStorePasswo

        GrpcUsersServerStub stub = new GrpcUsersServerSt

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFa
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostName(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

We will start by accessing the values of the default JVM properties (we will see how we configure these later) regarding both the filename that contains the server keystore and the password that protects that keystore.

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

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

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

        String keyStoreFilename = System.getProperty("javax.net.ssl.keyStore");
        String keyStorePassword = System.getProperty("javax.net.ssl.keyStorePassword");

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFac
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).bui

        String serverURI = String.format(SERVER_BASE_URI,

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

Then we obtain an instance of a keystore (initially empty) and load it with the cryptographic information stored in the server keystore. To do that we open a *FileInputStream* to the keystore file and load its contents providing the key that protects the file.

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

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

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

        String keyStoreFilename = System.getProperty("javax.net.ssl.keyStore");
        String keyStorePassword = System.getProperty("javax.net.ssl.keyStorePassword");

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFac
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).bui

        String serverURI = String.format(SERVER_BASE_URI,

        Log.info(String.format("Users gRPC Server ready
        server.start().awaitTermination();
    }
}
```

Next, we need a KeyManagerFactory that can be generated using its factory and using the default cryptographic algorithms.
We then load into this factory the keystore that we have previously prepared (again providing the password that protects that keystore).

gRPC Server and TLS

Class containing the main of the gRPC Server.

We can now initialize an SslContext that will leverage the KeyManagerFactory to be able to access the private key and public key certificate of the server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "gr

    private static Logger Log = Logger.getLogger(User

    public static void main(String[] args) throws Exc

        String keyStoreFilename = System.getProperty("javax.net.ssl.keyStore");
        String keyStorePassword = System.getProperty("javax.net.ssl.keyStorePassword");

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFactory)
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostName(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

gRPC Server and TLS

Class containing the main of the gRPC Server.

We can now use the NettyServerBuilder (instead of the Grpc class as we did in the first project) to create an instance of Server, providing the port of the server, the instance of the gRPC server stub, and the SSL Context that we have prepared.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "gr

    private static Logger Log = Logger.getLogger(User

    public static void main(String[] args) throws Exc

        String keyStoreFilename = System.getProperty('
        String keyStorePassword = System.getProperty('

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFactory)

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostName(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "gr

    private static Logger Log = Logger.getLogger(User

    public static void main(String[] args) throws Exc

        String keyStoreFilename = System.getProperty('
        String keyStorePassword = System.getProperty('

        KeyStore keystore = KeyStore.getInstance(KeyS

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFactory)
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostName(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

Similar to what we did in the REST server, the URL that the server announces for clients to connect to them can no longer have an IP address, instead it has to has the hostname of the machine where the server is running (and for which the private key and certificate were generated).

gRPC Server and TLS

Class containing the main of the gRPC Server.

```
package lab8.impl.server.grpc;

import java.io.FileInputStream;

@SuppressWarnings("unused")
public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

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

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

        String keyStoreFilename = System.getProperty("javax.net.ssl.keyStore");
        String keyStorePassword = System.getProperty("javax.net.ssl.keyStorePassword");

        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());

        try(FileInputStream input = new FileInputStream(keyStoreFilename)) {
            keystore.load(input, keyStorePassword.toCharArray());
        }

        KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
                KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keystore, keyStorePassword.toCharArray());

        GrpcUsersServerStub stub = new GrpcUsersServerStub();

        SslContext context = GrpcSslContexts.configure(
                SslContextBuilder.forServer(keyManagerFactory)
                ).build();

        Server server = NettyServerBuilder.forPort(PORT)
                .addService(stub).sslContext(context).build();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().g

                                                    ver ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

The server can now be started as we did before for a gRPC server.

EXECUTING THE SERVERS

When the servers execute, they must know what is the file that contains their keystore and the password for that keystore.

For clients, and for servers that also execute requests to other servers, we must also know the file containing the truststore file and its password.

This information is conveyed through arguments to the java machine:

```
java -Djavax.net.ssl.keyStore=<keystore-filename>
     -Djavax.net.ssl.keyStorePassword=<keystore-password>
     -Djavax.net.ssl.trustStore=<truststore-filename>
     -Djavax.net.ssl.trustStorePassword=<truststore-password>
<mainClass>
```

EXECUTING THE SERVERS

When the servers execute, they must know what is the file that contains their keystore and the password for that keystore.

For clients, and for servers that also execute requests to other servers, we must also know the file containing the truststore file and its password.

This information is conveyed through arguments to the java machine:

```
java -cp sd2425.jar -Djavax.net.ssl.keyStore=users-server.ks
     -Djavax.net.ssl.keyStorePassword=password
     -Djavax.net.ssl.trustStore=truststore.ks
     -Djavax.net.ssl.trustStorePassword=changeit lab8.impl.server.rest.UsersServer
```

This command is for the REST server in a docker named users.

EXECUTING THE SERVERS

When the servers execute, they must know what is the file that contains their keystore and the password for that keystore.

For clients, and for servers that also execute requests to other servers, we must also know the file containing the truststore file and its password.

This information is conveyed through arguments to the java machine:

```
java -cp sd2425.jar -Djavax.net.ssl.keyStore=users-server.ks
     -Djavax.net.ssl.keyStorePassword=password
     -Djavax.net.ssl.trustStore=truststore.ks
     -Djavax.net.ssl.trustStorePassword=changeit lab8.impl.server.grpc.UsersServer
```

This command is for the gRPC server in a docker named users.

EXECUTING THE SERVERS

The docker file that generates the image with your code must also be modified to copy the keystore and truststore files to the image:

```
# base ubuntu official image
FROM ubuntu

# run a command (install a package)
RUN apt-get update && apt-get install iproute2 -y

# Copy openjdk 17 from another image
ENV JAVA_HOME=/opt/java/openjdk
COPY --from=eclipse-temurin:17 $JAVA_HOME $JAVA_HOME
ENV PATH=$PATH:$JAVA_HOME/bin

# working directory inside docker image
WORKDIR /home/sd

# copy hibernate config
COPY hibernate.cfg.xml hibernate.cfg.xml

# copy keystore and truststore
COPY *.ks /home/sd/

# copy the jar created by assembly to the docker image
COPY target/*jar-with-dependencies.jar sd2425.jar

# run Discovery when starting the docker image
CMD ["java", "-cp", "sd2425.jar", "-Djavax.net.ssl.keyStore=/home/sd/users-server.ks" ,
                                  "-Djavax.net.ssl.keyStorePassword=password" ,
                                  "-Djavax.net.ssl.trustStore=/home/sd/truststore.ks" ,
                                  "-Djavax.net.ssl.trustStorePassword=changeit" ,
                                  "lab8.impl.server.grpc.UsersServer"]
```

EXECUTING THE SERVERS

The docker file that generates the image with your code must also be modified to copy the keystore and truststore files to the image:

```
# base ubuntu official image
FROM ubuntu

# run a command (install a package)
RUN apt-get update && apt-get install iproute2 -y

# Copy openjdk 17 from another image
ENV JAVA_HOME=/opt/java/openjdk
COPY --from=eclipse-temurin:17 $JAVA_HOME $JAVA_HOME
ENV PATH=$PATH:$JAVA_HOME/bin

# working directory inside docker image
WORKDIR /home/sd

# copy hibernate config
COPY hibernate.cfg.xml hibernate.cfg.xml

# copy keystore and truststore
COPY *.ks /home/sd/

# copy the jar created by assembly to the docker image
COPY target/*jar-with-dependencies.jar sd2425.jar

# run Discovery when starting the docker image
CMD ["java", "-cp", "sd2425.jar", "-Djavax.net.ssl.keyStore=/home/sd/users-server.ks" ,
                                  "-Djavax.net.ssl.keyStorePassword=password" ,
                                  "-Djavax.net.ssl.trustStore=/home/sd/truststore.ks" ,
                                  "-Djavax.net.ssl.trustStorePassword=changeit" ,
                                  "lab8.impl.server.grpc.UsersServer"]
```

(Notice that the example in this week executes the gRPC Server as default, also providing the jvm variables for the keystore and truststore)

GOALS

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

- Understand what is HTTPS and SSL/TLS
- Know how to generate a keystore with server cryptographic keys
- Know how to generate a truststore with root certificates and the certificate for your server
- Know how to develop a REST server using https in Java
- Know how to develop a gRPC server using TLS in Java
- **Know how to modify your REST clients to use https**
- **Know how to modify your gRPC clients to use TLS**

REST CLIENT CODE AND HTTPS

RestUsersClient Example

```
public class RestUsersClient extends UsersClient {
    private static Logger Log = Logger.getLogger(RestUsersClient.class.getName());

    final URI serverURI;
    final Client client;
    final ClientConfig config;

    final WebTarget target;

    public RestUsersClient( URI serverURI ) {
        this.serverURI = serverURI;

        this.config = new ClientConfig();

        config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
        config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);


        this.client = ClientBuilder.newClient(config);

        target = client.target( serverURI ).path( RestUsers.PATH );
    }
```

REST CLIENT CODE AND HTTPS

RestUsersClient Example

```
public class RestUsersClient extends UsersClient {
    private static Logger Log = Logger.getLogger(RestUsersClient.class.getName());

    final URI serverURI;
    final Client client;
    final ClientConfig config;

    final WebTarget target;

    public RestUsersClient( URI serverURI ) {
        this.serverURI = serverURI;

        this.config = new ClientConfig();

        config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
        config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);


        this.client = ClientBuilder.newClient(config);

        target = client.target( serverURI ).path( RestUsers.PATH );
    }
```

This shows the constructor of the RestUsersCliente (omitting the methods that effectively do the operations).

REST CLIENT CODE AND HTTPS

RestUsersClient Example

```
public class RestUsersClient extends UsersClient {
    private static Logger Log = Logger.getLogger(RestUsersClient.class.getName());

    final URI serverURI;
    final Client client;
    final ClientConfig config;

    final WebTarget target;

    public RestUsersClient( URI serverURI ) {
        this.serverURI = serverURI;

        this.config = new ClientConfig();

        config.property( ClientProperties.READ_TIMEOUT, READ_TIMEOUT);
        config.property( ClientProperties.CONNECT_TIMEOUT, CONNECT_TIMEOUT);


        this.client = ClientBuilder.newClient(config);

        target = client.target( serverURI ).path( RestUsers.PATH );
    }
```

This shows the constructor of the RestUsersClient (omitting the methods that effectively do the operations).

There are no changes here, it is sufficient that the URI that is used to initialize the client starts with https:// instead of http:// and that the server is identified by its hostname instead of IP address.

gRPC Client Code and TLS

gRPCUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslContexts
                .configure(
                        SslContextBuilder.forClient().trustManager(trustManagerFactory)
                        ).build();

        Channel channel = NettyChannelBuilder
                .forAddress(serverURI.getHost(), serverURI.getPort())
                .sslContext(context)
                .enableRetry()
                .build();
        stub = UsersGrpc.newBlockingStub( channel );
    }
```

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslContexts
                .configure(
                        SslContextBuilder.forClient().trustManager(trustManagerFactory)
                        ).build();
```

This shows the constructor of the GrpcUsersClient (omitting the methods that effectively do the operations).

```
    }
```

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslContexts
                .configure(
                        SslContextBuilder.forClient().trustManager(trustManagerFactory)
                        ).build();
```

This shows the constructor of the GrpcUsersClient (omitting the methods that effectively do the operations).

There are many more changes on this client that we have to see one by one.

```
    }
```

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI, String... args) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");

        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslCon
                .configure(
                        SslContextBuild
                        ).build();

        Channel channel = NettyChannelB
                .forAddress(serverURI.g
                .sslContext(context)
                .enableRetry()
                .build();
        stub = UsersGrpc.newBlockingStub( channel );
    }
```

We will start by accessing the values of the default JVM properties (that has shown before should be passed as an argument when starting the process) regarding both the filename that contains the truststore and the password that protects that truststore.

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");

        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustSt

        SslContext context = GrpcSslCont
                .configure(
                        SslContextBuilde
                        ).build();

        Channel channel = NettyChannelBu
                .forAddress(serverURI.ge
                .sslContext(context)
                .enableRetry()
                .build();
        stub = UsersGrpc.newBlockingStub
    }
```

Similar to the server side, we will have to create a (initially empty) truststore – *notice that a truststore is just a keystore, the main difference is that it only stores certificates with public keys of entities that we trust* – that we will load with the content of the file containing the truststore, also providing the password that protects that file.

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslContexts
                .configure(
                        SslContextBu
                        ).build();

        Channel channel = NettyChann
                .forAddress(serverUR
                .sslContext(context)
                .enableRetry()
                .build();
        stub = UsersGrpc.newBlocking
    }
```

Instead of a KeyManagerFactory, for the client side we instead need a TrustManagerFactory, that we can instanciate using its factory and using default cryptographic algorithms.
We then have to initialize this trustManagerFactory with the information that we loaded to our truststore.

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore
        try(FileInputStream input = ne
            trustStore.load(input, tru
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslContexts
                .configure(
                        SslContextBuilder.forClient().trustManager(trustManagerFactory)
                        ).build();

        Channel channel = NettyChannelBuilder
                .forAddress(serverURI.getHost(), serverURI.getPort())
                .sslContext(context)
                .enableRetry()
                .build();
        stub = UsersGrpc.newBlockingStub( channel );
    }
```

We can now generate a SslContext for clients providing to it the trustManagerFactory that we have just created and initialized.

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore.getInstance("JKS");
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory =
                TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SslContext context = GrpcSslContexts
                .configure(
                        SslContextBuilder.forClient().trustManager(trustManagerFactory)
                        ).build();

        Channel channel = NettyChannelBuilder
                .forAddress(serverURI.getHost(), serverURI.getPort())
                .sslContext(context)
                .enableRetry()
                .build();

    }
```

And now we can create the communication channel that we will be using the interact with the remote gRPC server. Notice that we are now using the NettyChannelBuilder (instead of the ManagedChannelBuilder as we were doing before). The options are the same (including the enableRetry option) but now we must provide also the SSL Context that contains the certificates with the public keys of entities that we trust.

gRPC Client Code and TLS

GrpcUsersClient Example

```
package lab8.clients.grpc;

import java.io.FileInputStream;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) throws Exception {
        String trustStoreFilename = System.getProperty("javax.net.ssl.trustStore");
        String trustStorePassword = System.getProperty("javax.net.ssl.trustStorePassword");


        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try(FileInputStream input = new FileInputStream(trustStoreFilename)) {
            trustStore.load(input, trustStorePassword.toCharArray());
        }

        TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance(
                TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStor

        SslContext context = GrpcSslContex
                .configure(
                        SslContextBuilder.forClient().trustManager(trustManagerFactory)
                        ).build();

        Channel channel = NettyChannelBuilder
                .forAddress(serverURI.getHost(), serverURI.getPort())
                .sslContext(context)
                .enableRetry()

        stub = UsersGrpc.newBlockingStub( channel );
```

Using the TLS enriched channel we can instantiate the client stub exactly as we did before.

EXECUTING THE CLIENTS

When the clients execute, they must know what is the file that contains the truststore and the password for that truststore.

Similar to the servers this information is conveyed through arguments to the java machine (but only those that refer to the truststore):

(it is similar for REST and gRPC)

```
Java -cp sd2425.jar -Djavax.net.ssl.trustStore=truststore.ks
      -Djavax.net.ssl.trustStorePassword=changeit lab8.clients.CreateUserClient
```

EXECUTING THE CLIENTS

When the clients execute, they must know what is the file that contains the truststore and the password for that truststore.

Similar to the servers this information is conveyed through arguments to the java machine (but only those that refer to the truststore):

(it is similar for REST and gRPC)

This is the example to execute the client that creates Users (it can interact with both REST and gRPC servers) in a docker container on the same network as the server.

```
Java -cp sd2425.jar -Djavax.net.ssl.trustStore=truststore.ks
     -Djavax.net.ssl.trustStorePassword=changeit lab8.clients.CreateUserClient
```

BONUS: REGARDING THE SECOND PROJECT

In the project your servers interact with each other, so you need each server to have a keystore and a truststore.

- You can use the same keystore for REST and gRPC servers.
- Servers must have both a keystore and a truststore (including all Root CA certificates and the certificate of the server itself)

BONUS: REGARDING THE SECOND PROJECT

Does using HTTPS makes your server secure?

BONUS: REGARDING THE SECOND PROJECT

Does using HTTPS makes your server secure?

Not exactly since clients can still execute the operations that are used by the server internally to do things as changing the posts of a user when the user account is deleted.

But using HTTPS allows us to enrich the security of those operations by adding a mandatory password that is shared among all servers to authenticate them. Since communication now is encrypted, these passwords will not be transmitted in cleartext.

EXERCISE (1/2)

In the code that supports this lab you will find an adapted version of the code from lab4 (with no information about the Image server) a keystore, truststore, (and the temporary certificate for the Users server). The cryptographic material was generated assuming the server will be executed in a containers named "users".

1. Use this code and cryptographic material to test both the REST and gRCP Users server with secure communication channels.
2. Launch the server container: `docker run -it --name users -h users --network sdnet sd2425-lab8-xxxxx-yyyyy /bin/bash`
3. In that container start one of the servers: e.g.,

```
java -cp sd2425.jar -Djavax.net.ssl.keyStore=users-server.ks -
Djavax.net.ssl.keyStorePassword=password
lab8.impl.server.grpc.UsersServer
```

4. Launch another container in the same network: `docker run -it --network sdnet sd2425-lab8-xxxxx-yyyyy /bin/bash`
5. In the second container execute the client with the correct URL: e.g.,

```
java -cp sd2425.jar -Djavax.net.ssl.trustStore=truststore.ks -
Djavax.net.ssl.trustStorePassword=changeit
lab8.clients.CreateUserClient grpc://users:9000/grpc jleitao
JoaoLeitao jc.leitao@fct.unl.pt password
```

EXERCISE (2/2)

- After validating that you can run the provided example in both REST and GRPC:

1. Delete the files providing the truststore, keystore, and server certificate (truststore.ks, users-server.ks, users.cert)
2. Use the instructions in the start of these slides to generate new keystore and truststore.
3. Verify that you can use your generated files to run the provided example (by generating again the docker image and going over the previous steps).
4. After this you can start modifying all servers (and clients) in your project to support secure communication.