Taller de Arquitectura Segura

Por: Luis Daniel Benavides Navarro

26-03-2020

Descripción

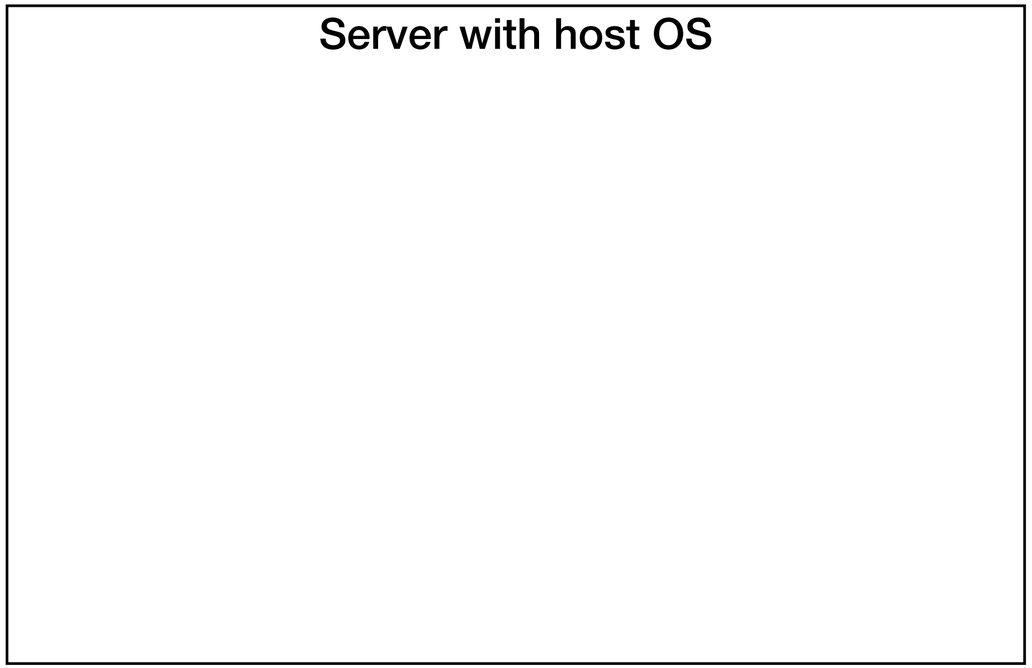
- Construir una aplicación web segura usando certificados digitales
- Pasos
  - Crear una aplicación web no segura
  - Crear un par de llaves y generar los certificados
  - Modificar la aplicación para que use los certificados
  - Explicar la esencia del la Arquitectura (Metáfora del sistema)
  - Asignar un reto de arquitectura

Herramientas

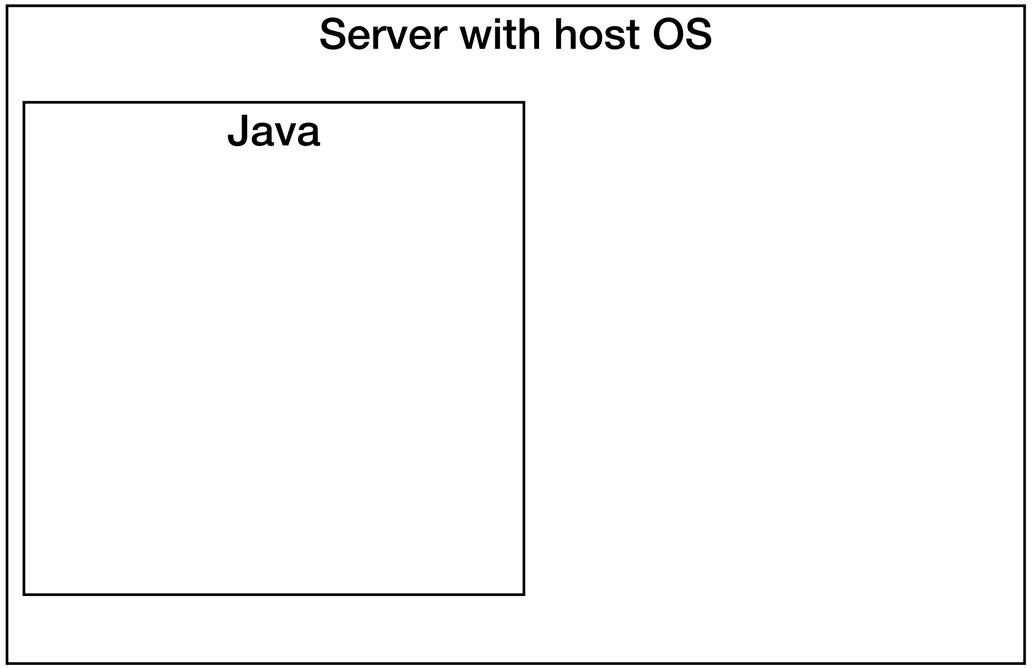
- **Java** como lenguaje de programación
- **Maven** para manejar el cíclo vida del proyecto
- **Spark Java.** “A micro framework for creating web applications in Kotlin and Java 8 with minimal effort”
- Una IDE poderosa (**Netbeans**)
- **Keytool** de java para generar y administrar las llaves y los certificados

Arquitectura del ejemplo

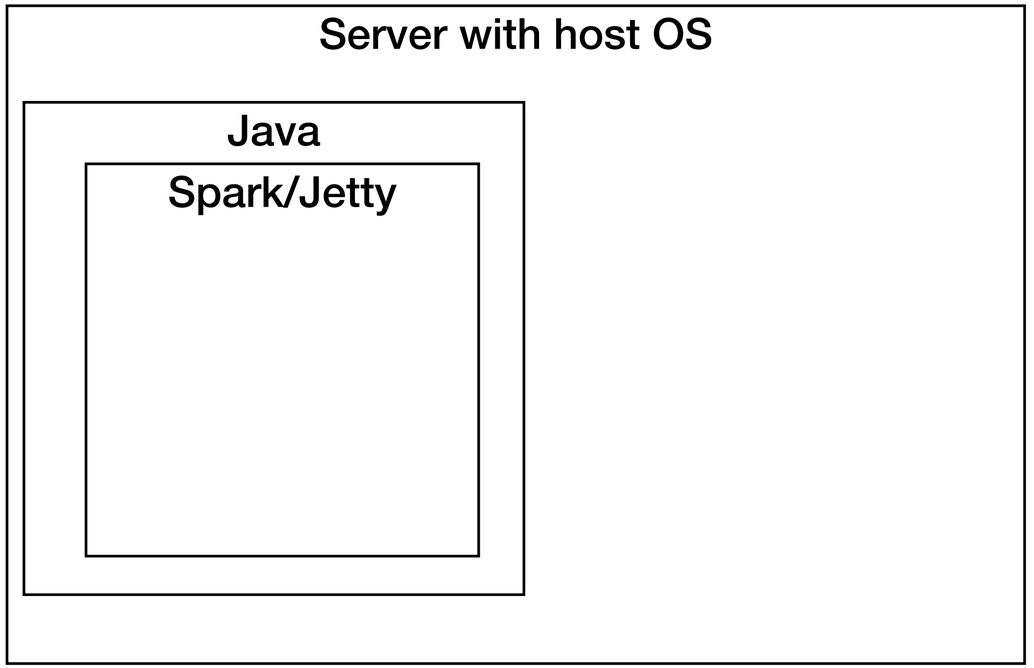
Arquitectura del ejemplo



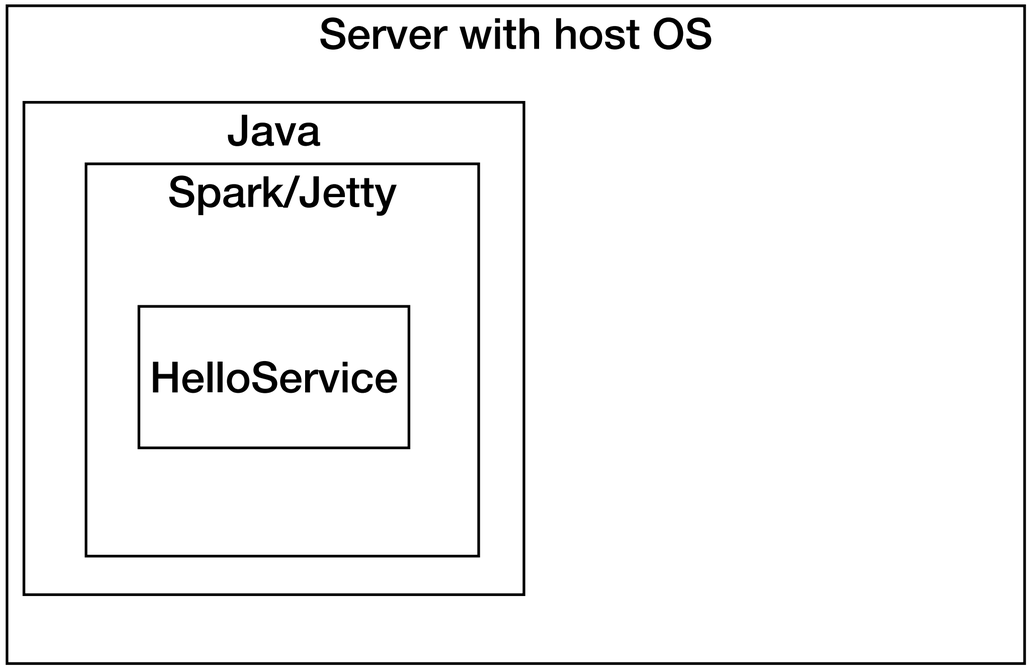
Arquitectura del ejemplo



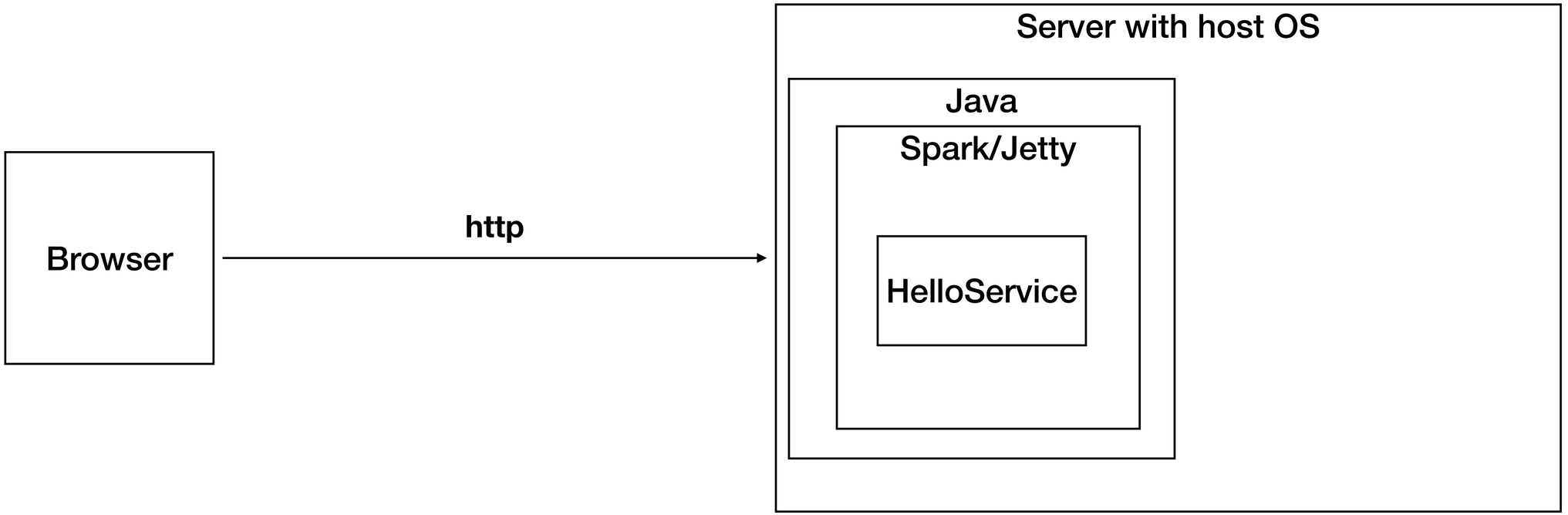
Arquitectura del ejemplo



Arquitectura del ejemplo



Arquitectura del ejemplo



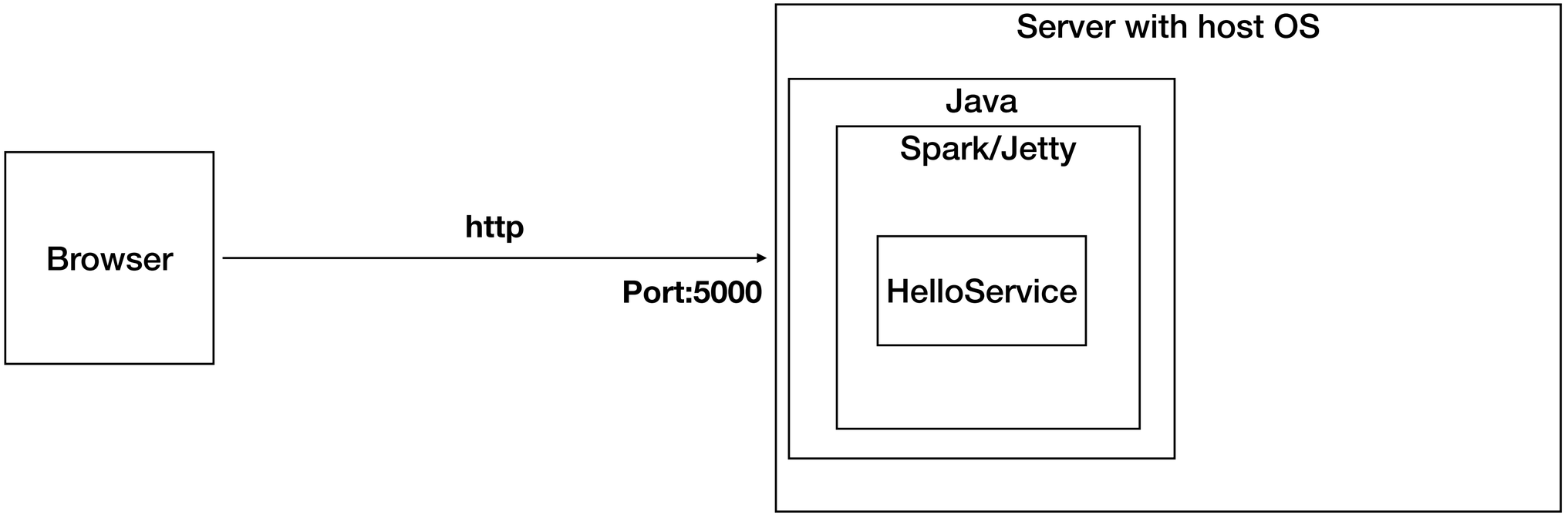
Implementación

- Sitio Web de Spark: http://sparkjava.com/

```
import static spark.Spark.*;

public class HelloWorld {

    public static void main(String[] args) {

        get("/hello", (req, res) -> "Hello World");

    }

}
```

```
Abrir en: http://localhost:4567/hello
```

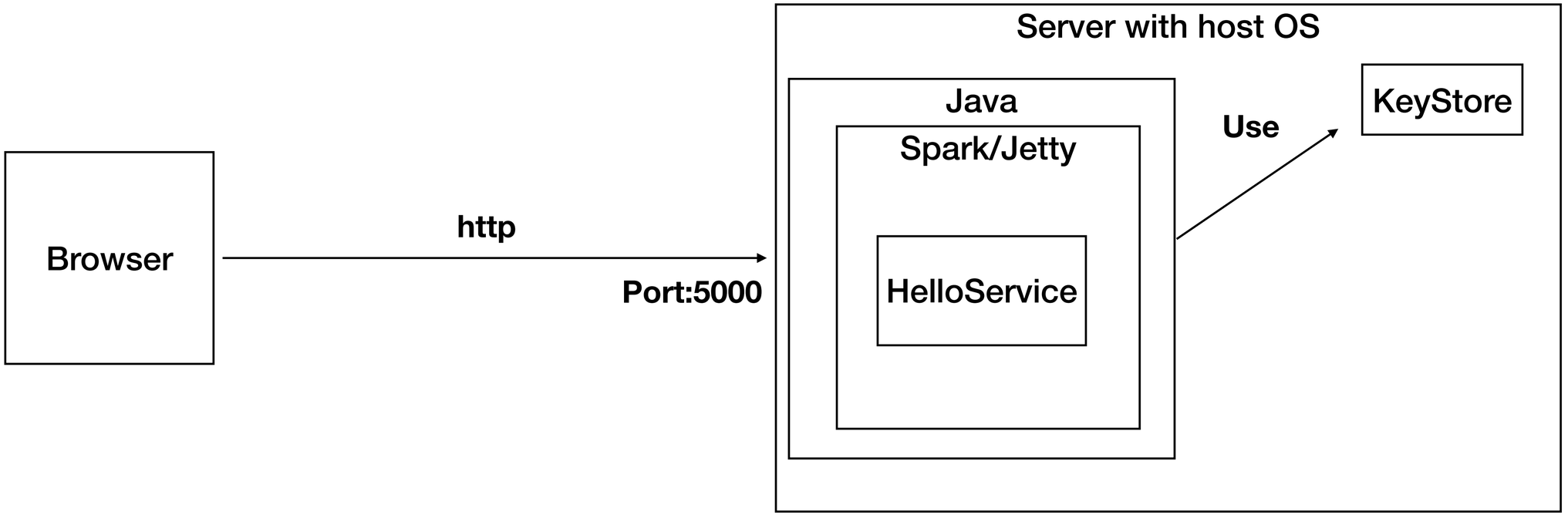
Arquitectura del ejemplo



Implementación

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

        port(5000);

        get("/hello", (req, res) -> "Hello Heroku");

    }
```

Arquitectura del ejemplo



Implementación

- Genere un par de llaves públicas y privadas y un certificado. Almacene todo en un archivo protegido.

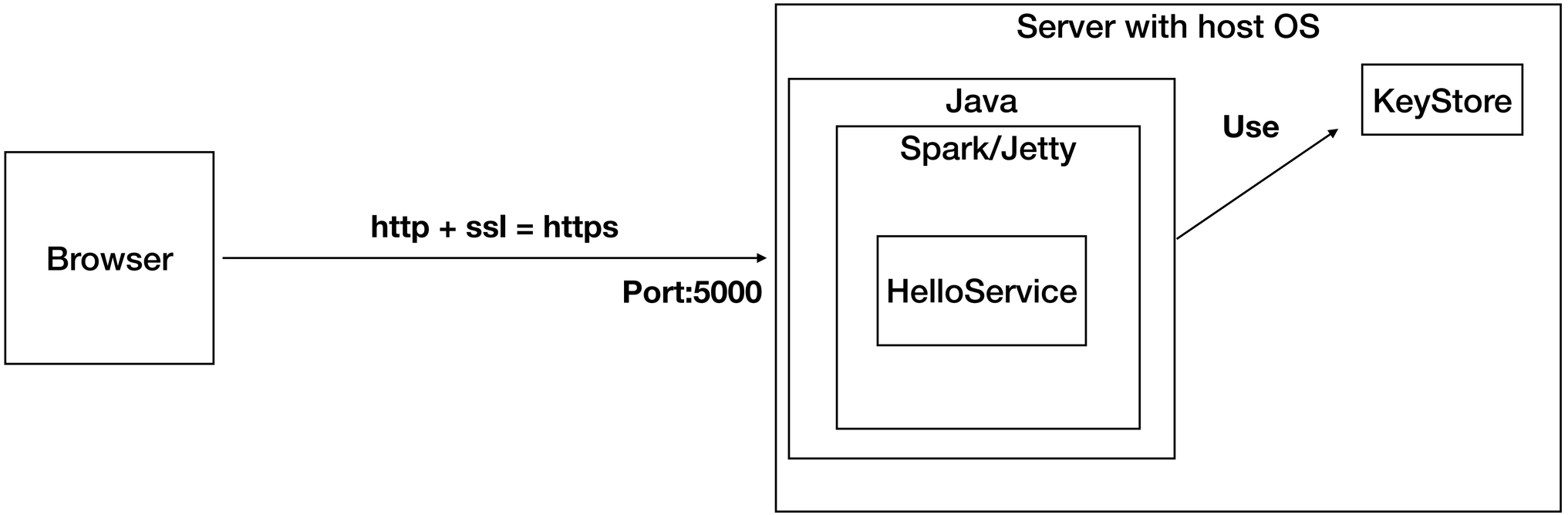
- Use el formato PKCS12 y no el JKS. La diferencia es que PKCS12 es un formato estándar para almacenar llaves y certificados, mientras que JKS es específico de Java.

- Comando:

```
keytool -genkeypair -alias ecikeypair -keyalg RSA -keysize 2048
-storetype PKCS12 -keystore ecikeystore.p12 -validity 3650
```

Nota: use “localhost” como nombre del certificado cuando se lo pida la herramienta.

Arquitectura del ejemplo



Implementación

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

 //API: secure(keystoreFilePath, keystorePassword, truststoreFilePath,
truststorePassword);

 secure("keystores/ecikeypair.p12", "ecistore", null, null);

 get("/hello", (req, res) -> "Hello World");

}
```

12 factor app

- Prepare su aplicación con las mejores prácticas
- 12 factor app.
- “**III. Config**

Store config in the environment

An app’s config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not. …”

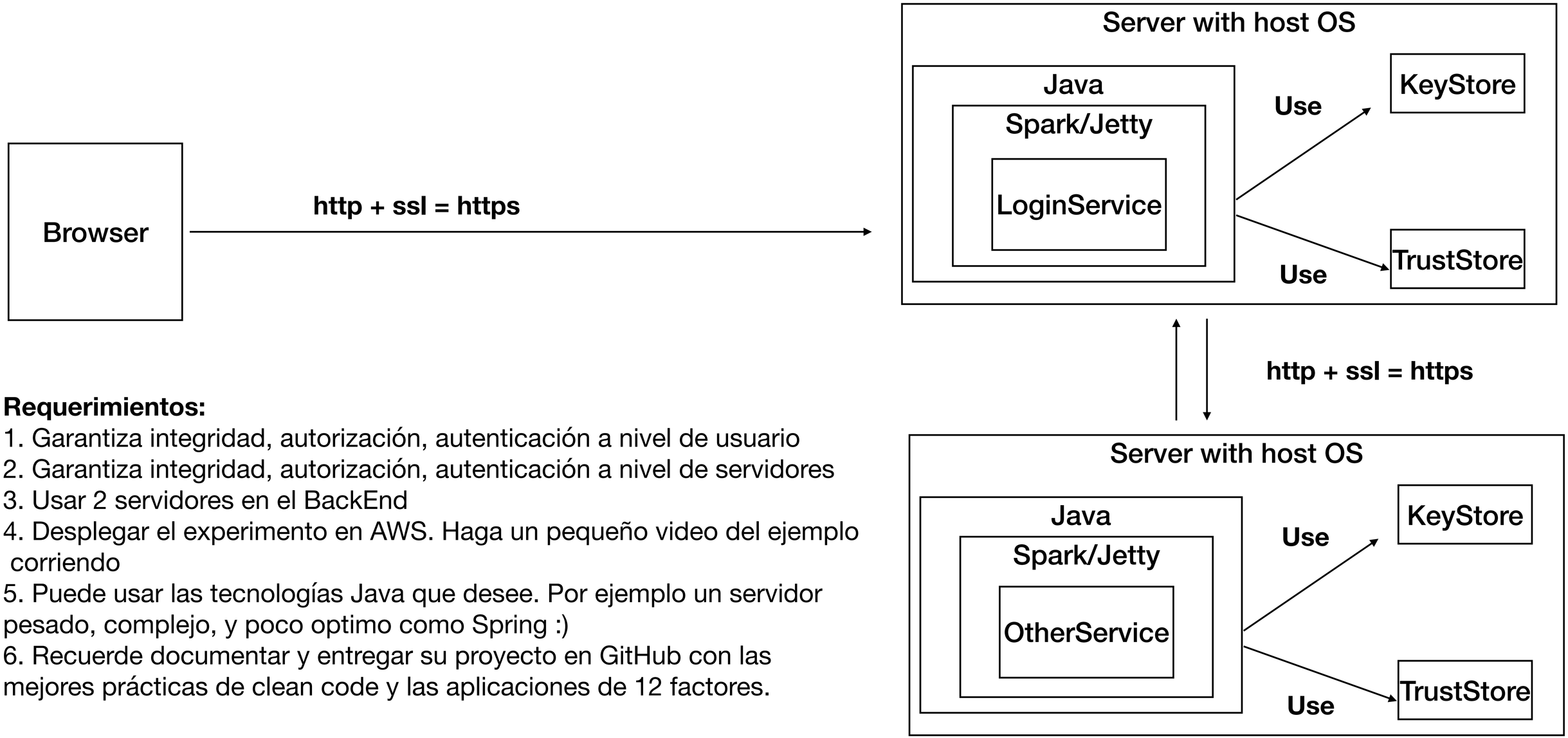
Implementación

- Ejemplo, lea el puerto desde el entorno

```
public static void main(String[] args) {
        port(getPort());
        get("/hello", (req, res) -> "Hello Heroku");
    }

    static int getPort() {
        if (System.getenv("PORT") != null) {
            return Integer.parseInt(System.getenv("PORT"));
        }
        return 5000; //returns default port if heroku-port isn't set (i.e. on localhost)
    }
```

- Puede hacer lo mismo con el keystone?

Reto: construya una aplicación Segura



Requerimientos:

1. Garantiza integridad, autorización, autenticación a nivel de usuario
2. Garantiza integridad, autorización, autenticación a nivel de servidores
3. Usar 2 servidores en el BackEnd
4. Desplegar el experimento en AWS. Haga un pequeño video del ejemplo corriendo
5. Puede usar las tecnologías Java que desee. Por ejemplo un servidor pesado, complejo, y poco optimo como Spring :)
6. Recuerde documentar y entregar su proyecto en GitHub con las mejores prácticas de clean code y las aplicaciones de 12 factores.

Exporte el certificado a un archivo

```
keytool -export -keystore ./ecikeystore.p12 -alias ecikeypair -file ecicert.cer
```

Importe el certificado a un TrustStore

```
keytool -import -file ./ecicert.cer -alias firstCA -keystore myTrustStore
```

Implemente un SecureUrlReader

```
// Create a file and a password representation
File trustStoreFile = new File("keystores/myTrustStore");
char[] trustStorePassword = "567890".toCharArray();

// Load the trust store, the default type is "pkcs12", the alternative is "jks"
KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
trustStore.load(new FileInputStream(trustStoreFile), trustStorePassword);

// Get the singleton instance of the TrustManagerFactory
TrustManagerFactory tmf = TrustManagerFactory
        .getInstance(TrustManagerFactory.getDefaultAlgorithm());

// Itit the TrustManagerFactory using the truststore object
tmf.init(trustStore);
```

Implemente un SecureUrlReader II

```
//Set the default global SSLContext so all the connections will use it
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
SSLContext.setDefault(sslContext);

// We can now read this URL
readURL("https://localhost:5000/hello");

// This one can't be read because the Java default truststore has been
// changed.
readURL("https://www.google.com");
```

Fin