

TEMA 8

SERVICIOS WEB, AJAX Y RICH INTERNET APPLICATIONS

APLICACIONES WEB - GIS - CURSO 2017/18



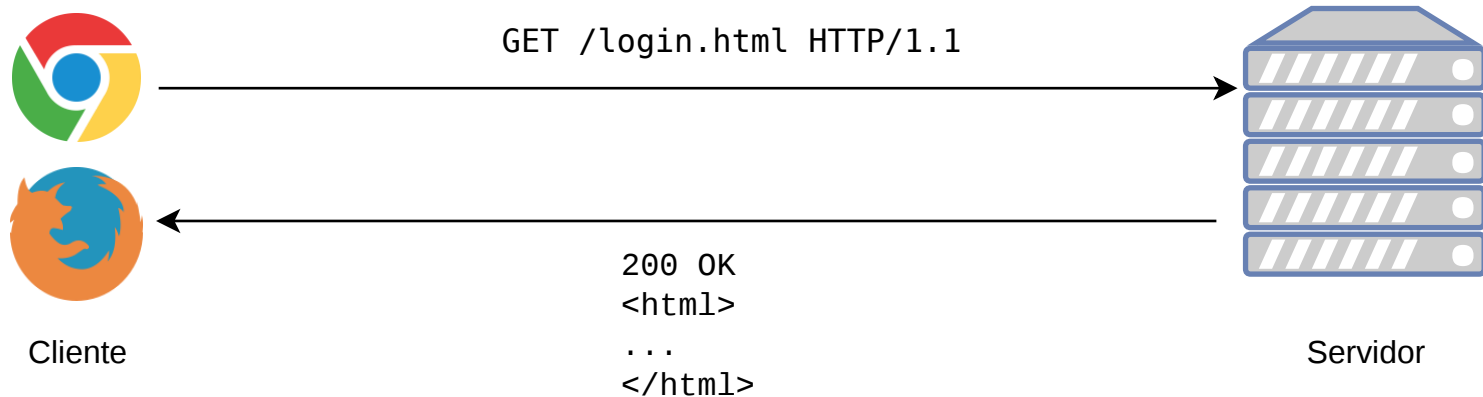
Esta obra está bajo una
Licencia CC BY-NC-SA 4.0 Internacional.

Manuel Montenegro [montenegro@fdi.ucm.es]
Dpto de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

1. **INTRODUCCIÓN**
2. **FORMATOS XML Y JSON**
3. **SERVICIOS REST**
4. **PETICIONES AJAX**
5. **AUTENTICACIÓN BÁSICA**
6. **INTRODUCCIÓN A SOAP**
7. **EL PROTOCOLO HTTPS**
8. **BIBLIOGRAFÍA**

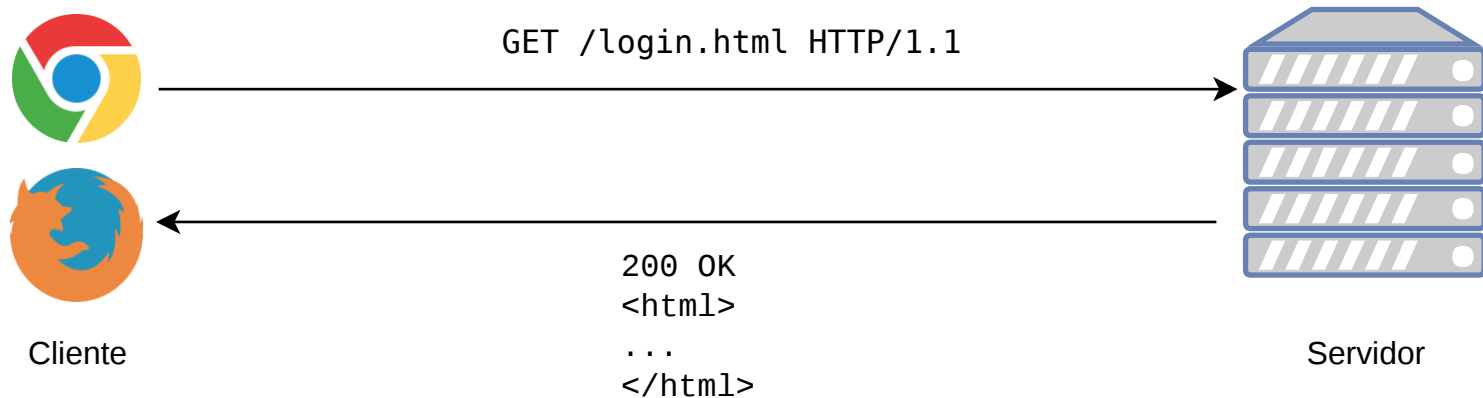


INTRODUCCIÓN



Hasta ahora hemos estudiado el protocolo HTTP utilizado en la comunicación entre un navegador web y un servidor.

INTRODUCCIÓN



El servidor devolvía al cliente: documentos HTML (**.html**), imágenes (**.png**, **.jpg**, **.svg**), código Javascript para ser ejecutado en navegador (**.js**), etc.

El destinatario de estos contenidos es el **navegador web**

INTRODUCCIÓN



¿Y si utilizamos el mismo esquema y el mismo protocolo HTTP para que el servidor devuelva otro tipo de contenido?

El contenido estaría destinado a ser consumido por un programa que tome el papel del cliente; no necesariamente un navegador.

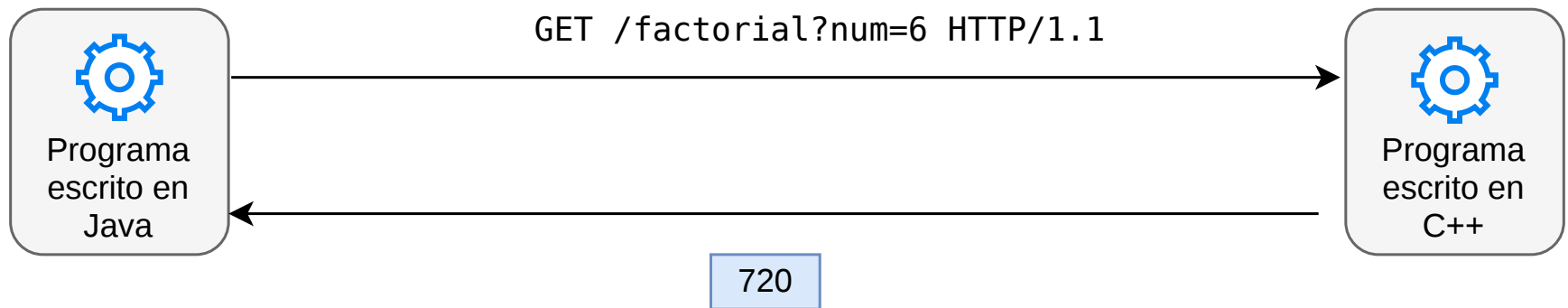
SERVICIOS WEB

Un **servicio web** es un servicio ofrecido por un dispositivo a otros dispositivos a través de la web.

Los servicios web están diseñados para la comunicación máquina a máquina.

La comunicación requerida por el servicio se realiza normalmente mediante HTTP.

La principal ventaja es la **interoperabilidad** entre distintos componentes, independientemente del lenguaje de programación en el que estén implementados y de la máquina en la que se ejecuten.



EJEMPLOS DE SERVICIOS WEB

- **Fixer.io** - <http://fixer.io/>
Proporciona una API para los tipos de cambio de divisas.
- **OpenWeatherMap** - <https://openweathermap.org/>
Permite acceder a los datos meteorológicos de distintas ciudades.
- **Google Drive, Dropbox, GitHub**
Disponen de una API mediante la cual pueden implementarse aplicaciones que accedan, desde el escritorio o de manera automática, a ficheros almacenados en la nube.

PILA DE PROTOCOLOS DE SERVICIOS WEB

- **Protocolo de transporte.** Determina el mecanismo de envío de mensajes. Normalmente es HTTP.
- **Protocolo de mensajería.** Determina cómo se codifican los mensajes. Suele utilizarse XML (SOAP) o JSON.
- **Protocolo de descripción.** Describe la interfaz de un servicio (qué operaciones se pueden realizar, qué parámetros recibe cada una, etc.). Por ejemplo, WSDL.
- **Protocolo de descubrimiento.** Permite que un servicio publique su URL y descripción en un registro público común. Por ejemplo, UDDI.

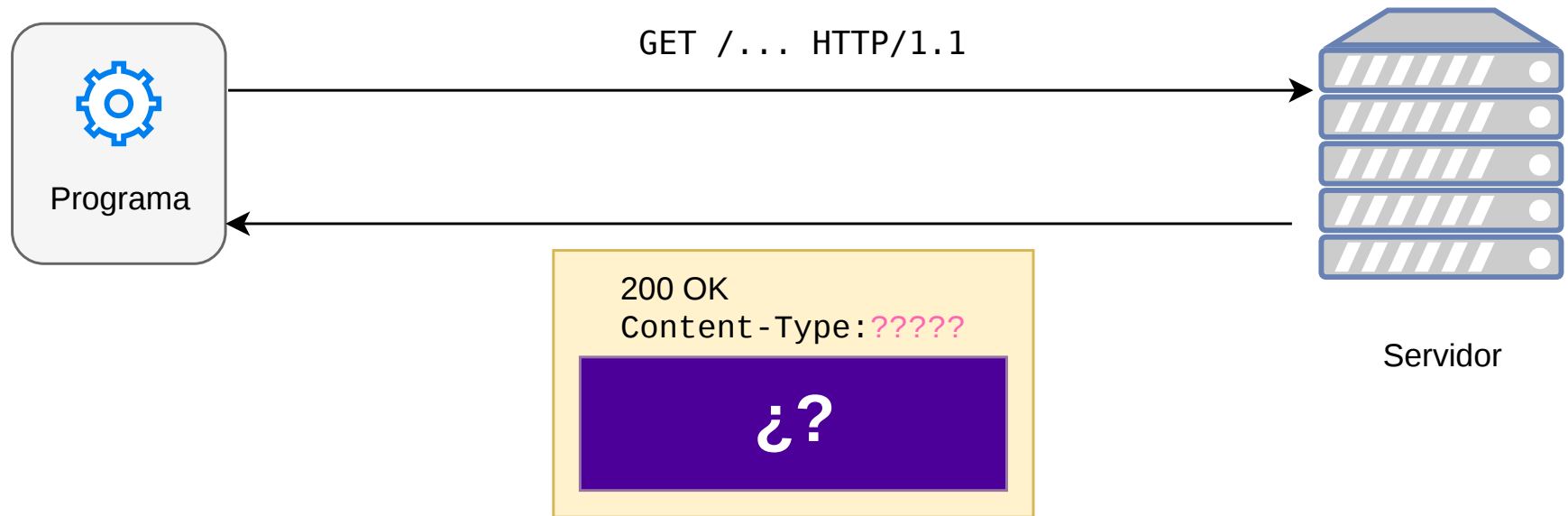
TIPOS DE SERVICIOS WEB

- **REST** (*REpresentational State Transfer*)
Utilizan los distintos tipos de peticiones HTTP (**GET**, **POST**, **PUT**, **DELETE**) para especificar las operaciones a realizar. Las operaciones se realizan sobre recursos, cada uno identificado mediante una URL.
- **SOAP** (*Simple Object Access Protocol*)
Utilizan mensajes en formato XML para especificar el tipo de operación que se desea realizar y sus parámetros.

1. INTRODUCCIÓN
2. **FORMATOS XML Y JSON**
3. SERVICIOS REST
4. PETICIONES AJAX
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. EL PROTOCOLO HTTPS
8. BIBLIOGRAFÍA



FORMATOS DE INTERCAMBIO DE DATOS



Hemos visto que el servidor responde al programa cliente con la información que este último ha pedido.
¿En qué formato se transmite esta información?

REQUISITOS DE UN FORMATO DE INTERCAMBIO DE DATOS

- Debe ser lo suficientemente **genérico** como para poder representar cualquier tipo de respuesta.
- Debe de ser fácilmente analizable por un programa.
... y si es legible, mejor.

Formatos de intercambio más populares:

- **XML** (*eXtensible Markup Language*)
- **JSON** (*JavaScript Object Notation*)

Nosotros utilizaremos principalmente JSON.

FORMATO XML

```
<agenda>

  <contacto id="c1">
    <nombre>David Jiménez</nombre>
    <fecha-nacimiento>20/01/1989</fecha-nacimiento>
    <telefonos>
      <tel tipo="casa">917371235</tel>
      <tel tipo="trabajo">914483124</tel>
    </telefonos>
  </contacto>

  <contacto id="c2">
    <nombre>Luis García</nombre>
    <fecha-nacimiento>15/03/1979</fecha-nacimiento>
    <telefonos>
    </telefonos>
  </contacto>

</agenda>
```

Sintaxis similar a la de HTML, con ciertas restricciones:

- Toda etiqueta de inicio tiene una etiqueta de cierre, y las etiquetas han de estar correctamente anidadas.
- Los atributos de etiquetas están delimitados por comillas dobles.
- Existe un único elemento raíz.

Cualquier documento que cumpla estas tres condiciones
está **bien formado**.

El conjunto de etiquetas XML permitidas y su contenido depende de la aplicación concreta.

En nuestro ejemplo: `<agenda>`, `<contacto>`, `<nombre>`, `<fecha-nacimiento>`, `<telefonos>` y `<tel>`.

Existen mecanismos para especificar restricciones adicionales: qué etiquetas y atributos se permiten, y cuál es el contenido de cada una de ellas.

Si un documento XML bien formado cumple estas restricciones, se dice que es un documento **válido**.

LENGUAJES DE ESPECIFICACIÓN PARA XML

- DTD (*Document Type Definition*)

```
<!ELEMENT agenda (contacto*)>
<!ELEMENT contacto (nombre, fecha-nacimiento?, telefonos)>
<!ELEMENT telefonos (tel*)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT fecha-nacimiento (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
<!ATTLIST contacto id ID #REQUIRED>
<!ATTLIST tel tipo CDATA #REQUIRED>
```

- XML Schema
- Relax NG

OTRAS APLICACIONES DE XML

- **Ficheros de configuración**

Por ejemplo: *Ant*, *Maven*, *Eclipse*, *Tomcat*, etc.

- **Bases de datos**

Por ejemplo: *BaseX*, *eXist-db*, etc.

Existe un lenguaje de consultas propio: *XQuery*.

FORMATO JSON

```
{  
  "c1": {  
    "nombre": "David Jiménez",  
    "fechaNacimiento": "20/01/1989",  
    "telefonos": [  
      { "tipo": "casa", "numero": 917371235 },  
      { "tipo": "trabajo", "numero": 914483124 }  
    ]  
  },  
  "c2": {  
    "nombre": "Luis García",  
    "fechaNacimiento": "15/03/1979",  
    "telefonos" : []  
  }  
}
```

Sintaxis similar a los objetos de Javascript

... con una excepción: los nombres de los atributos van siempre delimitados por comillas

Incorrecto:

```
{ nombre: "Lucía", edad: 20 }
```

Correcto:

```
{ "nombre": "Lucía", "edad": 20 }
```

SINTAXIS JSON

- **Números**

```
34    2.40    -2.0e+16
```

- **Cadenas**

```
"Hola"    "Esto es una cadena"
```

- **Booleanos:** `true`, `false`

- Valor nulo: `null`

- **Arrays**

```
[ 23, "Pepe", 3.45, null, 2 ]
```

- **Objetos** (listas de pares clave-valor)

```
{ "nombre" : "Javier", "asignaturas": ["MDLM", "ABD", "MMI"] }
```

EJEMPLO

Supongamos un servicio web de conversión de divisas.

Petición:

```
{  
  "moneda": "EUR",  
  "cantidad": 25.00,  
  "convertirA": ["GBP", "JYN", "USD"]  
}
```

Respuesta:

```
[  
  { "moneda": "GBP", "cantidad": 21.3226 },  
  { "moneda": "JYN", "cantidad": 3064.69 },  
  { "moneda": "USD", "cantidad": 26.1096 }  
]
```

LENGUAJES DE ESPECIFICACIÓN PARA JSON

Permiten especificar la estructura que ha de seguir un determinado fichero JSON.

La especificación más conocida es JSON Schema:

<http://json-schema.org/>

Sin embargo, apenas se utiliza.

UTILIZAR JSON EN JAVASCRIPT

Las siguientes funciones están disponibles tanto en el servidor (Node) como en el cliente (navegadores).

- **JSON.parse(*texto*)**

Convierte un texto en formato JSON en un objeto Javascript

- **JSON.stringify(*valor*)**

Obtiene la representación JSON de un objeto Javascript.

```
var obj = [ {x: 13, y: 12}, {x: 0, y: 0} ];
JSON.stringify(obj);
// → Devuelve la cadena ' [{"x":13,"y":12}, {"x":0,"y":0}] '

JSON.parse(' [{"x":13,"y":12}, {"x":0,"y":0}] ');
// → Devuelve el objeto [ {x: 13, y: 12}, {x: 0, y: 0} ]
```


UTILIZAR JSON EN JAVA

Existe una multitud de librerías externas.

Por ejemplo: **org.json**

<https://github.com/stleary/JSON-java>

```
String cadena = "[{\"x\":13,\"y\":12},{\"x\":0,\"y\":0}]";  
  
JSONArray arr = new JSONArray(cadena);  
  
System.out.println(arr.getJSONObject(0).get("x")); // → 13  
System.out.println(arr.getJSONObject(0).get("y")); // → 12
```

1. INTRODUCCIÓN
2. FORMATOS XML Y JSON
3. **SERVICIOS REST**
4. PETICIONES AJAX
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. EL PROTOCOLO HTTPS
8. BIBLIOGRAFÍA



SERVICIOS DE TIPO REST

Utilizan la semántica de los métodos HTTP (**GET**, **POST**, etc.) para indicar la acción a realizar sobre un determinado recurso, identificado por una URL.

Principios básicos:

1. Todo es un recurso.
2. Los recursos están identificados por URIs.
3. Se utilizan los verbos HTTP estándar.
4. Un recurso puede representarse de múltiples formas.
5. La comunicación con los servicios web no tiene estado.

RECURSOS Y URIs

Cada recurso tiene una URI que lo identifica.

Ejemplos:

- /contactos/elisa
- /contactos/elisa/telefonos/casa
- /contactos
- /calculadora/factorial

Las URIs no hacen referencia necesariamente a un fichero físico en el servidor, sino a un recurso «abstracto»

MÉTODOS HTTP

Suelen utilizarse las acciones **GET**, **POST**, **PUT** y **DELETE**, que se corresponden con las operaciones CRUD habituales.

CRUD = *Create, Read, Update, Delete*

Método HTTP	Semántica
GET	Leer un recurso
POST	Crear un nuevo recurso
PUT	Actualizar un recurso existente
DELETE	Eliminar un recurso

EJEMPLOS

- Acceder al contacto cuyo identificador es **elisa**
GET /contactos/elisa
- Calcular factorial
GET /calculadora/factorial?num=6
- Añadir un nuevo contacto
POST /contactos
- Actualizar el contacto con identificador **elisa**
PUT /contactos/elisa

La descripción semántica de estos recursos es «orientativa»

Nada nos impide utilizar un método para realizar una acción distinta a la especificada por su semántica.

No obstante, algunos métodos imponen de manera implícita ciertas propiedades:

- El método **GET** no altera el estado del recurso al que hace referencia.
- Los métodos **GET**, **PUT** y **DELETE** son idempotentes. Realizar una misma operación varias veces seguidas es equivalente a realizarla una única vez.

CÓDIGOS DE RESPUESTA

Método	Código de respuesta
GET	200 OK - Tuvo éxito
	404 Not Found - Recurso no encontrado
	500 Internal Server Error - Otros errores
POST	201 Created - Tuvo éxito
	500 Internal Server Error - Otros
PUT	200 OK - Tuvo éxito
	500 Internal Server Error - Otros
DELETE	200 OK, 204 No Content - Tuvo éxito
	404 Not Found - Recurso no existe
	500 Internal Server Error - Otros

REPRESENTACIONES DE UN RECURSO

Un recurso puede ser representado de distintas formas.

Representación JSON:

```
{  
  "id": "elisa",  
  "nombre": "Elisa Trujillo",  
  "telefonos": [{"casa": "912382722"}]  
}
```

Representación XML:

```
<contacto id="elisa">  
  <nombre>Elisa Trujillo</nombre>  
  <telefonos>  
    <tel tipo="casa">912382722</tel>  
  </telefonos>  
</contacto>
```

En las peticiones de tipo **GET** el cliente especifica en la **cabecera** el tipo de representación que desea recibir:

```
Accept: application/json
```

```
Accept: text/xml
```

En las peticiones de tipo **PUT** o **DELETE**, el cliente especifica en la **cabecera** de la petición la representación del nuevo objeto.

```
Content-Type: application/json
```

```
Content-Type: text/xml
```

AUSENCIA DE ESTADO

En los servicios web de tipo REST se respeta la carencia de estado propia del protocolo HTTP.

No debe hacerse uso de los mecanismos para mantener variables de estado que perviven entre peticiones: cookies, sesiones, etc.

IMPLEMENTACIÓN DE SERVICIOS WEB DE TIPO REST CON EXPRESS.JS

Se puede utilizar el framework *Express.js* para implementar servicios web de tipo REST.

Supongamos un objeto de la clase *Application*:

```
var app = express();
```

Ya conocemos los métodos *app.get()* y *app.post()* para definir funciones que manejan las peticiones GET y POST.

- *app.get(url, funcionManejadora)*
- *app.post(url, funcionManejadora)*

Existen sendos métodos `put()` y `delete()` para manejar las peticiones PUT y DELETE

- `app.put(url, funcionManejadora)`
- `app.delete(url, funcionManejadora)`

Al igual que en los métodos `get()` y `post()`, es posible añadir middleware específico en cada petición:

```
app.put("/contactos/:id",  
        middleware_1, middleware_2,  
        function(request, response) {  
            ...  
        }  
);
```

Middleware intermedio

EJEMPLO: AGENDA DE CONTACTOS

Supongamos que tenemos un array con objetos. Cada uno de ellos contiene un nombre y un número de teléfono:

```
var agenda = [  
  { nombre: "Pepita", telefono: "89731982" },  
  { nombre: "Puri", telefono: "28329828" },  
  { nombre: "David", telefono: "827272728" }  
];
```

Diseñamos un servicio web de tipo REST para que puedan realizarse las siguientes funciones:

- Obtener todos los objetos.
- Obtener un objeto a partir de su índice.
- Añadir un objeto.
- Eliminar un objeto.
- Actualizar un objeto.

1. OBTENER TODOS LOS OBJETOS

Método	GET
URL	/contactos
Parám. entrada	Ninguno
Códigos respuesta	200 si tuvo éxito, 500 en caso de error.
Tipos resultado	JSON
Resultado	Lista de objetos. Cada objeto tiene dos atributos: nombre y telefono.

IMPLEMENTACIÓN

El método `response.json()` permite devolver una respuesta en formato JSON.

Recibe un objeto Javascript que transformará en JSON.

Es una operación terminal. Llama a `end()` automáticamente y, tras realizarse, no puede escribirse nada más en el objeto `response`.

En nuestro caso nos limitamos a devolver el array `agenda`.

```
app.get("/contactos", function(request, response) {  
  // Por defecto se devuelve el código 200, por  
  // lo que no hace falta indicarlo mediante el  
  // método response.status().  
  response.json(agenda);  
});
```

2. OBTENER UN OBJETO A PARTIR DE SU ÍNDICE

Método	GET
URL	<i>/contactos/:indice</i>
Parám. entrada	Índice del objeto a recuperar (desde 0 hasta la longitud del array menos uno), incluido en la URL.
Códigos respuesta	200 si tuvo éxito, 404 si no se proporciona un índice válido, y 500 en otro caso de error.
Tipos resultado	JSON
Resultado	Un objeto con dos atributos: <i>nombre</i> y <i>telefono</i> .

Existen varias formas de pasar información como parámetro a un servicio:

- Mediante **URLs paramétricas**
 - Por ejemplo: `/contactos/:indice`
 - Se acceden mediante `request.params`
 - Por ejemplo, al acceder a la URL `/contactos/2`, la variable `request.params.indice` toma el valor "2".
- Mediante **query strings**
 - Por ejemplo: `/contactos?ind=2`
 - Se acceden mediante `request.query`
 - Por ejemplo `request.query.indice` vale "2".
- En el **cuerpo** de la petición.
 - Mediante el middleware `body-parser`.
 - Sólo válido en peticiones **POST** y **PUT**.

En nuestro caso hemos optado por una URL paramétrica:

```
app.get("/contactos/:indice", function(request, response) {  
  var indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined) {  
    // Si el número está dentro de los límites del array  
    // devolvemos el elemento correspondiente  
    var elem = agenda[indice];  
    response.json(elem);  
  } else {  
    // En caso contrario, devolvemos un error 404  
    response.status(404);  
    response.end();  
  }  
});
```

Si quisiéramos pasar el índice mediante *query strings* (p. ej. `/contacto?ind=2`) utilizaríamos `request.query`:

```
app.get("/contacto", function(request, response) {  
  var indice = Number(request.query.ind);  
  ...  
  // igual que antes  
  ...  
});
```

¿Qué pasaría si en lugar de la URL `/contacto` utilizásemos `/contactos` para este tipo de peticiones?

```
app.get("/contactos", function(request, response) {  
  var indice = Number(request.query.ind);  
  ...  
});
```

3. AÑADIR UN OBJETO

Método	POST
URL	/contactos
Parám. entrada	JSON con el objeto a insertar.
Códigos respuesta	201 si tuvo éxito, y 500 en caso de error.
Tipos resultado	Ninguno
Resultado	Ninguno

¿Cómo pasamos un JSON como parámetro a la petición?

El JSON debe ir en el **cuerpo** de la misma.

TIPOS DE CUERPO DE PETICIÓN

Hasta ahora conocemos dos:

- `application/x-www-form-urlencoded`

Opción por defecto en formularios. El contenido podía ser accedido mediante el middleware `body-parser`.

```
var bodyParser = require("body-parser");  
...  
app.use(bodyParser.urlencoded({extended: false}));
```

- `multipart/form-data`

Utilizado en aquellos formularios que adjuntan ficheros. En este caso utilizábamos el middleware `multer`.

Ahora introducimos un tercer tipo:

- `application/json`

Utilizado para pasar información a servicios web que soporten JSON. En este caso también podemos utilizar el middleware `body-parser`.

```
var bodyParser = require("body-parser");  
...  
app.use(bodyParser.json());
```

El método `bodyParser.json()` devuelve un middleware que convierte el JSON contenido en el cuerpo del mensaje en un objeto Javascript, y lo asigna al atributo `request.body`.

Con este middleware podemos implementar la operación de inserción de elementos al final del array:

```
app.use(bodyParser.json());  
// ...  
  
// Añadir contactos  
app.post("/contactos", function(request, response) {  
  var nuevoElemento = request.body;  
  agenda.push(nuevoElemento);  
  response.status(201);  
  response.end();  
});
```

4. ELIMINAR UN OBJETO

Método	DELETE
URL	<i>/contactos/:indice</i>
Parám. entrada	Índice del elemento a eliminar. Viene incluido en la URL.
Códigos respuesta	200 si tuvo éxito, 404 si el índice no es correcto, y 500 en otro caso de error.
Tipos resultado	Ninguno
Resultado	Ninguno

IMPLEMENTACIÓN

```
app.delete("/contactos/:indice", function(request, response) {  
  var indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined) {  
    agenda.splice(indice, 1);  
    // Código 200 = OK  
    response.status(200);  
  } else {  
    // Error 404 = Not found  
    response.status(404);  
  }  
  response.end();  
});
```

5. ACTUALIZAR UN OBJETO

Método	PUT
URL	<i>/contactos/:indice</i>
Parám. entrada	Índice del elemento a actualizar. Viene incluido en la URL. Nuevo valor del elemento (objeto con atributos nombre y telefono) dentro del cuerpo del mensaje, en formato JSON.
Códigos respuesta	200 si tuvo éxito, 404 si el índice no es correcto, y 500 en otro caso de error.
Tipos resultado	Ninguno
Resultado	Ninguno

IMPLEMENTACIÓN

```
app.put("/contactos/:indice", function(request, response) {  
  var indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined) {  
    agenda[indice] = request.body;  
  } else {  
    // Error 404 = Not Found  
    response.status(404);  
  }  
  response.end();  
});
```

COMPROBAR EL FUNCIONAMIENTO DE UN SERVICIO WEB

Existen varias herramientas que permiten al usuario realizar cualquier tipo de peticiones HTTP a un servidor.

Una de las más básicas es **curl** en GNU/Linux.

```
# curl -i -X GET "http://localhost:3000/contactos/1"
```

devuelve:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 39
ETag: W/"27-QvXARR27S+g6IfKZZLdKUQ"
Date: Wed, 28 Dec 2016 17:16:10 GMT
Connection: keep-alive
```

```
{"nombre":"Puri","telefono":"28329828"}
```

No obstante, hay aplicaciones más completas y más intuitivas. Algunas de ellas se instalan como extensiones (*plug-ins*) en el navegador web.

- En Firefox: **RESTClient**
<http://restclient.net/>
- En Chrome: **Advanced REST Client**
<https://advancedrestclient.com/>

PETICIONES GET EN ADVANCED REST CLIENT

Peticiones GET con Advanced REST Client



PETICIONES POST EN ADVANCED REST CLIENT

Peticiones POST con Advanced REST Client



ENVÍO DE FICHEROS BINARIOS

Cuando se quiere realizar una petición **POST** o **PUT** adjuntando un fichero binario (no de texto) en el cuerpo de la petición, el formato JSON resulta inadecuado.

En este caso, es mejor adjuntar a la petición un cuerpo con el tipo **multipart/form-data**.

Los detalles de implementación son los mismos que en las aplicaciones vistas en el Tema 6. Para ello es necesario el uso del middleware **multer**.

EJEMPLO

Servicio para subir una imagen:

```
var multer = require("multer");
var upload = multer({ storage: multer.memoryStorage() });
...

app.put("/contactos/:indice/imagen", upload.single("imagen"),
  function(request, response) {
    var indice = Number(request.params.indice);
    if (!isNaN(indice) && agenda[indice] !== undefined) {
      // Creamos un nuevo atributo llamado 'imagen' al objeto
      // correspondiente.
      agenda[indice].imagen = request.file.buffer;
    } else {
      response.status(404);
    }
    response.end();
  });
```

Servicio para obtener la imagen de un contacto:

```
app.get("/contactos/:indice/imagen", function(request, response) {  
  var indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined  
      && agenda[indice].imagen !== undefined) {  
    response.end(agenda[indice].imagen, 'binary');  
  } else {  
    response.status(404);  
    response.end();  
  }  
});
```

SUBIR UN FICHERO BINARIO CON ADVANCED REST CLIENT

Envío de ficheros mediante Advanced REST Client



1. INTRODUCCIÓN
2. FORMATOS XML Y JSON
3. SERVICIOS REST
4. **PETICIONES AJAX**
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. EL PROTOCOLO HTTPS
8. BIBLIOGRAFÍA



PETICIONES AJAX

Hasta ahora hemos contemplado tres formas en las que el navegador realiza una petición HTTP:

- El usuario introduce una URL en la barra de direcciones del navegador. Este último envía una petición **GET** con la URL introducida.
- El usuario hace clic en un enlace. El navegador realiza una petición **GET** con la URL indicada en el atributo **href**.
- El usuario hace clic en el botón *Submit* del formulario. El navegador realiza una petición **GET** o **POST** (según el atributo **method** del formulario) a la URL indicada en el atributo **action** del formulario.

Existe otro mecanismo mediante el cual es posible realizar peticiones HTTP mediante **Javascript** en el navegador

AJAX = *Asynchronous Javascript and XML*

Anteriormente se utilizaba el formato XML para pasar parámetros a las peticiones AJAX, y para recibir los resultados.

Nosotros utilizaremos JSON.

Existen diferencias entre los distintos navegadores sobre cómo realizar peticiones AJAX.

- Según el estándar W3C: clase `XMLHttpRequest`.
- En Internet Explorer versión ≤ 6 : control ActiveX `Microsoft.XMLHTTP`.

Nosotros realizaremos las peticiones AJAX mediante la librería *jQuery*, que abstrae todas estas diferencias en una única función: `$.ajax()`

LA FUNCIÓN \$.ajax()

`$.ajax(opciones)`

El parámetro *opciones* es un objeto que especifica los detalles de la petición. Contiene los siguientes atributos:

- **method**

Método HTTP a utilizar: GET, POST, PUT, DELETE, etc.

- **url**

URL sobre la que realizar la petición.

- **success, error**

Funciones *callback* llamadas cuando la petición ha tenido éxito o ha fallado, respectivamente.

- **data**

Datos a adjuntar en la petición, ya sea como parámetros de la *query string* (esto es, detrás de la URL en el caso de peticiones GET), o dentro del cuerpo de la petición (en el caso de peticiones POST o PUT).

- **contentType**

Tipo MIME del cuerpo de la petición. Por defecto es **application/x-www-form-urlencoded**.

FUNCIÓN CALLBACK: **success**

Cuando el servidor responde la petición con un código de éxito (**2xx**), se llamará a esta función con tres argumentos:

```
$.ajax({ ...  
  success: function(data, textStatus, jqXHR) {  
    ...  
  }  
});
```

- **data**: Datos contenidos en el cuerpo de la respuesta HTTP. Si el formato de la respuesta es JSON, se convierte automáticamente en un objeto Javascript.
- **statusText**: Cadena que describe el estado de la petición (normalmente "**success**").
- **jqXHR**: Objeto con más información sobre la respuesta (envoltorio del objeto **XMLHttpRequest**).

FUNCIÓN CALLBACK: **error**

```
$.ajax({ ...  
  error: function(jqXHR, textStatus, errorThrown) {  
    ...  
  }  
});
```

- **jqXHR**: Envoltorio del objeto **XMLHttpRequest**.
- **statusText**: Cadena que describe el estado de la petición ("**timeout**", "**parseerror**", "**abort**", etc.)
- **errorThrown**: Texto adjunto a la respuesta HTTP devuelta ("**Not Found**", "**Internal Server Error**", etc.).

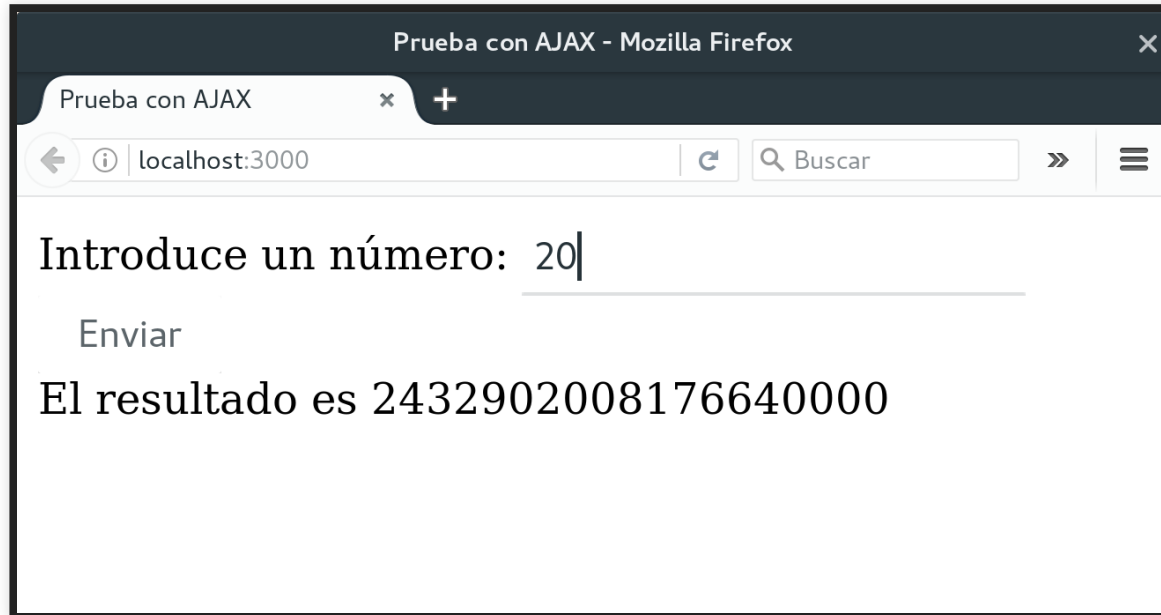
EJEMPLO

Supongamos un servicio web para calcular el factorial de un número dado:

Método	GET
URL	<code>/factorial/:num</code>
Parám. entrada	Número cuyo factorial se desea calcular. Viene incluido en la URL paramétrica (parámetro <code>num</code>)
Códigos respuesta	200 si tuvo éxito, 400 (<i>Bad Request</i>) si no se ha introducido un número, y 500 en otro caso de error.
Tipos resultado	JSON
Resultado	Un objeto <code>{"result": fact}</code> , donde <i>fact</i> es el factorial calculado.

```
app.get("/factorial/:num", function(request, response) {  
  var numero = Number(request.params.num);  
  if (!isNaN(numero) && numero >= 0) {  
    // Calculamos el factorial  
    var f = 1;  
    for (var i = 2; i <= num; i++) {  
      f *= i;  
    }  
  
    // Devolvemos el resultado  
    response.json({ resultado: f });  
  } else {  
    response.status(400);  
    response.end();  
  }  
});
```

Realizamos llamadas a este servicio desde una página Web:



```
<div>
  <label for="cuadroTexto">Introduce un número: </label>
  <input type="text" id="cuadroTexto">
</div>
<button id="botonEnviar">Enviar</button>
<div id="resultado"></div> <div>Aquí se mostrará el resultado</div>
```


Código Javascript:

```
$(document).ready(function() {  
    // Cada vez que se pulse el botón de 'Enviar'  
    $("#botonEnviar").on("click", function() {  
        // Obtenemos el valor contenido en el cuadro de texto  
        var valor = $("#cuadroTexto").val();  
        // Realizamos la petición al servidor  
        $.ajax({  
            type: "GET",  
            url: "/factorial/ " + valor,  
            // En caso de éxito, colocamos el texto con el resultado  
            // en el documento HTML  
            success: function (data, textStatus, jqXHR) {  
                console.log(textStatus);  
                $("#resultado").text(  
                    "El resultado es " + data.resultado);  
            },  
            // En caso de error, mostramos el error producido  
            error: function (jqXHR, textStatus, errorThrown) {  
                alert("Se ha producido un error: " + errorThrown);  
            }  
        });  
    });  
});
```

PETICIONES GET CON PARÁMETROS

En este ejemplo hemos considerado que la URL era paramétrica:

Por ejemplo: `/factorial/5`

Ahora supongamos que el número viene dado como parámetro en la petición GET:

Por ejemplo: `/factorial?num=5`

Código del servidor:

```
app.get("/factorial", function(request, response) {  
  var numero = Number(request.query.num);  
  // ...  
  // igual que antes  
  // ...  
});
```

Código en el navegador:

```
$.ajax({  
  type: "GET",  
  url: "/factorial",  
  
  // Parámetros de la petición. Como la petición es de tipo GET,  
  // se pasarán como cadena al final de la URL. Por ejemplo:  
  // /factorial?num=5  
  data: {  
    num: valor  
  },  
  
  // ... igual que antes ...  
});
```

Cuando se pasa un objeto Javascript a la opción **data**, este se transforma en una *query string*.

Por ejemplo, **{key1: val1, ..., keyn: valn}** se transforma en **key1=val1&key2=val2&...&keyn=valn**

PETICIONES CON JSON

Ahora suponemos que el número cuyo factorial se desea calcular viene dado en el **cuerpo** de la petición, en forma de un JSON:

```
{ "num": 6 }
```

Con una petición de tipo GET no podemos adjuntar un cuerpo, por lo que utilizamos una petición POST.

En el lado del servidor necesitamos utilizar el middleware **body-parser**, que interpreta el JSON contenido en el cuerpo del mensaje y añade el objeto correspondiente a la propiedad **request.body**:

```
app.use(bodyParser.json());  
// ...  
  
app.post("/factorial", function(request, response) {  
    var numero = Number(request.body.num);  
    // ... igual que antes ...  
});
```

En el lado del cliente, debemos adjuntar la cadena JSON en la propiedad **data**. Para ello utilizamos el método **JSON.stringify()**.

Por otro lado, indicamos mediante **contentType** el tipo MIME de la información enviada en el cuerpo del mensaje.

```
$.ajax({  
  type: "POST",  
  url: "/factorial",  
  contentType: "application/json",  
  data: JSON.stringify({ num: valor }),  
  // ...  
});
```

1. INTRODUCCIÓN
2. FORMATOS XML Y JSON
3. SERVICIOS REST
4. PETICIONES AJAX
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. EL PROTOCOLO HTTPS
8. BIBLIOGRAFÍA



AUTENTICACIÓN BÁSICA

A menudo es necesario restringir el uso de un servicio web a un determinado conjunto de usuarios.

Mediante un sistema de autenticación, cualquier usuario que quiera acceder deberá proporcionar su identidad, generalmente mediante un nombre de usuario y contraseña.

El protocolo HTTP incorpora mecanismos de control de acceso autenticado.

ESQUEMA DE AUTENTICACIÓN HTTP

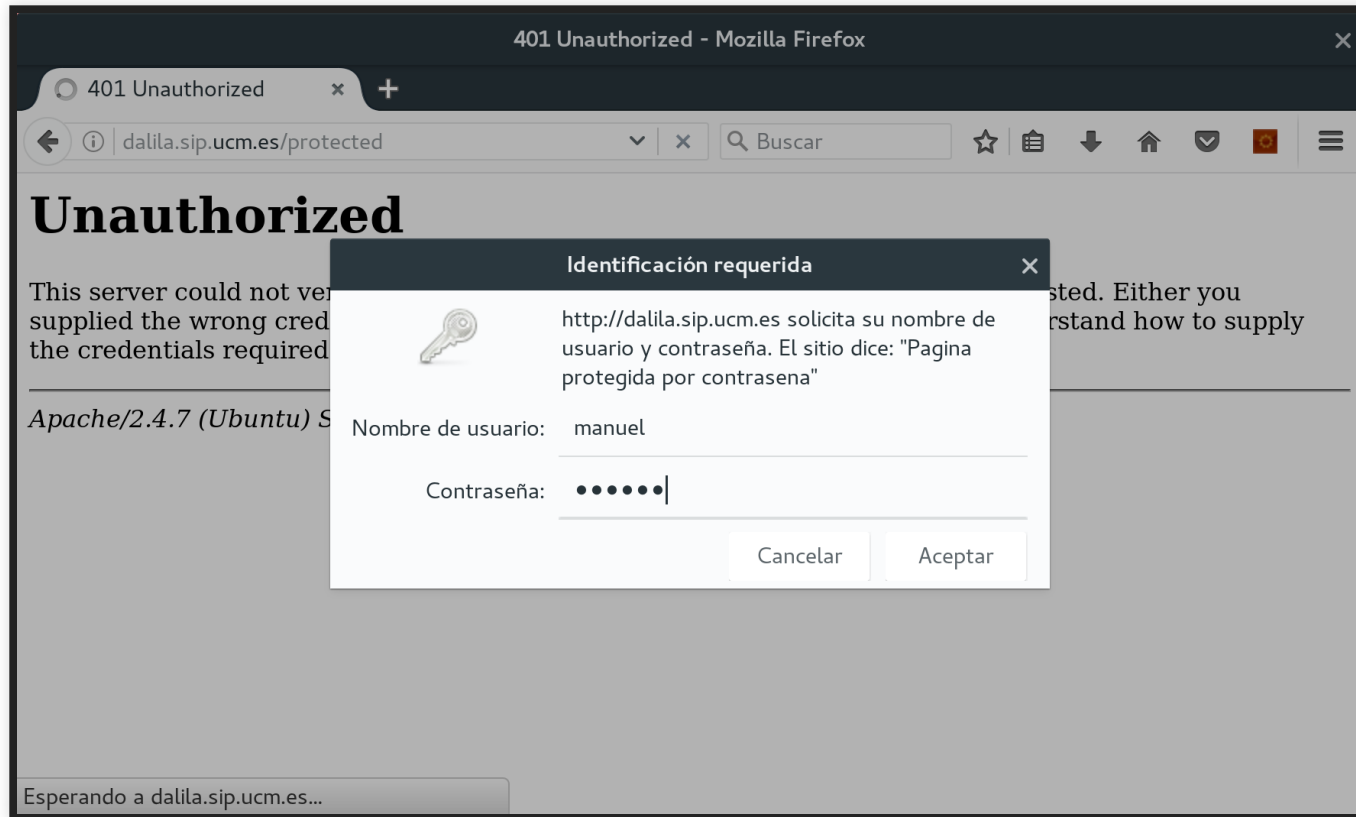
Supongamos que un navegador web quiere acceder a un recurso que requiere autenticación.

Paso 1. El servidor devuelve un código 401 (**Unauthorized**) indicando, en las cabeceras de la respuesta, el **reino de autenticación**.

```
HTTP/1.1 401 Unauthorized
...
WWW-Authenticate: Basic realm="Pagina protegida por contraseña"
...
```

Todos los recursos pertenecientes al mismo reino comparten el mismo sistema de autenticación.

Paso 2. El navegador, cuando recibe un error 401, muestra una ventana al usuario invitándole a introducir los datos de identificación: nombre de usuario y contraseña.



Paso 3. El navegador construye una cadena con el usuario y contraseña proporcionados, utilizando `:` como separador.

En nuestro ejemplo, `"manuel:123456"`

La cadena se codifica en base 64: `bWFudWVsOjEyMzQ1Ng==`

Paso 4. El navegador vuelve a realizar la misma petición al recurso protegido, pero añadiendo una cabecera adicional con la cadena de autenticación codificada:

```
Authorization: Basic bWFudWVsOjEyMzQ1Ng==
```

Paso 4.1. Si la autenticación tiene éxito, se procesa la petición y se devuelve un código 200 (**OK**) con el recurso correspondiente.

Paso 4.2. Si la autenticación es fallida, se vuelve a devolver un error 401 (**Unauthorized**).

Hemos presentado la cabecera **Authentication**. ¿Qué hacemos con ella?

Si queremos incorporar un sistema de autenticación en nuestros servicios web:

- **En el servidor:** debemos reconocer y procesar la cabecera **Authorization** de las peticiones HTTP.
- **En el cliente:** debemos incluir la cabecera **Authorization** en las peticiones AJAX.

AUTENTICACIÓN BÁSICA EN EL SERVIDOR

Existe un paquete en Node (llamado `passport`) que permite gestionar la autenticación utilizando distintos mecanismos.

Nosotros utilizaremos el mecanismo de autenticación básico, que se encuentra en el paquete `passport-http`.

```
npm install --save passport  
npm install --save passport-http
```

Importamos los módulos `passport` y `passport-http`:

```
var passport = require("passport");  
var passportHTTP = require("passport-http");
```

El método `passport.initialize()` devuelve un middleware que debe incluirse en la cadena de middlewares antes de realizar algún tipo de autenticación.

```
app.use(passport.initialize());
```


Se denomina **estrategia** a cada mecanismo de autenticación incorporado a una aplicación web.

El método **passport.use()** permite incorporar una estrategia nueva al catálogo de estrategias utilizadas por la aplicación.

Nosotros incorporaremos una estrategia básica (clase **BasicStrategy**), definida en el módulo **passportHTTP**.

```
var miEstrategia =  
  new passportHTTP.BasicStrategy({ realm: "...", /* funcion */});  
passport.use(miEstrategia);
```

La función pasada como parámetro al constructor de **BasicStrategy** recibe tres parámetros:

- El nombre de usuario **user** proporcionado por el cliente en la autenticación.
- La contraseña **pass** proporcionada por el cliente.
- Una función callback **f**.

```
var miEstrategia =  
  new passportHTTP.BasicStrategy(  
    { realm: "Pagina protegida" },  
    function(user, pass, f) {  
      // Comprobar datos de autenticación  
    }  
  );
```

La función de autenticación, que debemos implementar nosotros, comprobará que los datos son correctos.

- Si se produce error, debe llamar a `f(err)`.
- Si no se produce error, pero los datos introducidos son incorrectos, se debe llamar a `f(null, false)`.
- Si los datos son correctos, se llamará a `f(null, user)`, donde `user` es un objeto que contendrá los datos del usuario identificado.
 - El middleware de autenticación guardará este objeto en el atributo `request.user`.


En nuestro ejemplo, supongamos que requerimos que el nombre de usuario sea **manuel** y la contraseña **123456**:

```
passport.use(new passportHTTP.BasicStrategy(  
  { realm: 'Autenticacion requerida' },  
  function(user, pass, callback) {  
    if (user === "manuel" && pass === "123456") {  
      callback(null, { userId: "Manuel" });  
    } else {  
      callback(null, false);  
    }  
  }  
));
```

Una vez incorporado el mecanismo de identificación básico en `passport`, el método `passport.authenticate()` devuelve un middleware que realizará la comprobación correspondiente.

Este middleware debe incorporarse en aquellos recursos en los que la autenticación sea necesaria:

```
app.get("/protegido",  
  passport.authenticate('basic', {session: false}),  
  function(request, response) {  
    response.json({permitido: true});  
  });
```



La opción `{session: false}` indica que la autenticación no involucra ni *cookies* ni sesiones.

ACCESO AUTENTICADO MEDIANTE EL CLIENTE REST

Autenticación mediante Advanced REST Client



AUTENTICACIÓN BÁSICA MEDIANTE AJAX

Hemos de añadir la siguiente cabecera a la petición HTTP:

```
Authorization: Basic bWFudWVsOjEyMzQ1Ng==
```

Para ello:

1. Codificamos la cadena *usuario:contraseña* en base 64, utilizando la función *btoa*.

```
btoa("manuel:123456") // → "bWFudWVsOjEyMzQ1Ng=="
```

2. Añadimos la cabecera utilizando la opción *beforeSend* de la función *\$.ajax()*.

EJEMPLO

```
$(document).ready(function() {  
    var user = "manuel";  
    var pass = "123456";  
  
    var cadenaBase64 = btoa(user + ":" + pass);  
  
    $.ajax({  
        method: "GET",  
        url: "/protegido",  
        beforeSend: function(req) {  
            // Añadimos la cabecera 'Authorization' con los datos  
            // de autenticación.  
            req.setRequestHeader("Authorization",  
                                "Basic " + cadenaBase64);  
        },  
        success: function(data, state, jqXHR) {  
            if (data.permitted) {  
                console.log("¡Acceso permitido!");  
            }  
        }  
    });  
});
```


1. INTRODUCCIÓN
2. FORMATOS XML Y JSON
3. SERVICIOS REST
4. PETICIONES AJAX
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. EL PROTOCOLO HTTPS
8. BIBLIOGRAFÍA



SERVICIOS DE TIPO SOAP

A diferencia de los servicios de tipo REST, los servicios SOAP no están basados en el acceso a recursos mediante el uso de métodos HTTP.

Esencialmente hacen uso de peticiones POST y GET, utilizando XML como mecanismo de comunicación.

Para cada servicio, existe un fichero (**WSDL**) que especifica el catálogo de operaciones ofrecidas por el servicio.

EJEMPLO: TABLA PERIÓDICA

<http://www.webservicex.net/periodictable.asmx?WSDL>

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions ... >
  ...
  <wsdl:message name="GetAtomicNumberHttpGetIn">
    <wsdl:part name="ElementName" type="s:string" />
  </wsdl:message>
  <wsdl:message name="GetAtomicNumberHttpGetOut">
    <wsdl:part name="Body" element="tns:string" />
  </wsdl:message>
  ...
  <wsdl:operation name="GetAtomicNumber">
    <wsdl:documentation>
      Get atomic Number by element name
    </wsdl:documentation>
    <wsdl:input message="tns:GetAtomicNumberHttpGetIn" />
    <wsdl:output message="tns:GetAtomicNumberHttpGetOut" />
  </wsdl:operation>
  ...
</wsdl:service>
</wsdl:definitions>
```

El fichero WSDL indica, entre otras cosas, que existe un método llamado **GetAtomicNumber** que recibe una cadena con el nombre de un elemento de la tabla periódica, y devuelve una cadena con información sobre ese elemento.

ACCESO A SERVICIOS SOAP DESDE JAVA

Java dispone de una herramienta que genera clases de acceso a servicios web a partir del fichero WSDL.

```
# wsimport http://www.webserviceex.net/periodictable.asmx?WSDL
```

Esta herramienta es accesible desde Netbeans:

File → New File... → Web Service → Web Service Client

En este caso se genera, entre otras, una clase llamada **Periodictable**, que proporciona acceso a los servicios.

EJEMPLO

```
public static void main(String[] args) {  
    Periodictable pt = new Periodictable();  
    String datos =  
        pt.getPeriodictableSoap().getAtomicNumber("Oxygen");  
    System.out.println(datos);  
}
```

Resultado:

```
<NewDataSet>  
  <Table>  
    <AtomicNumber>8</AtomicNumber>  
    <ElementName>Oxygen</ElementName>  
    <Symbol>O</Symbol>  
    <AtomicWeight>15.9994</AtomicWeight>  
    <BoilingPoint>90.2</BoilingPoint>  
    <IonisationPotential>13.61</IonisationPotential>  
    <ElectroNegativity>3.5</ElectroNegativity>  
    <AtomicRadius>0.74</AtomicRadius>  
    <MeltingPoint>55</MeltingPoint>  
    <Density>1.3318</Density>  
  </Table>  
</NewDataSet>
```

Podemos utilizar las herramientas de análisis XML incorporadas en las librerías de Java para extraer los datos del texto XML devuelto.

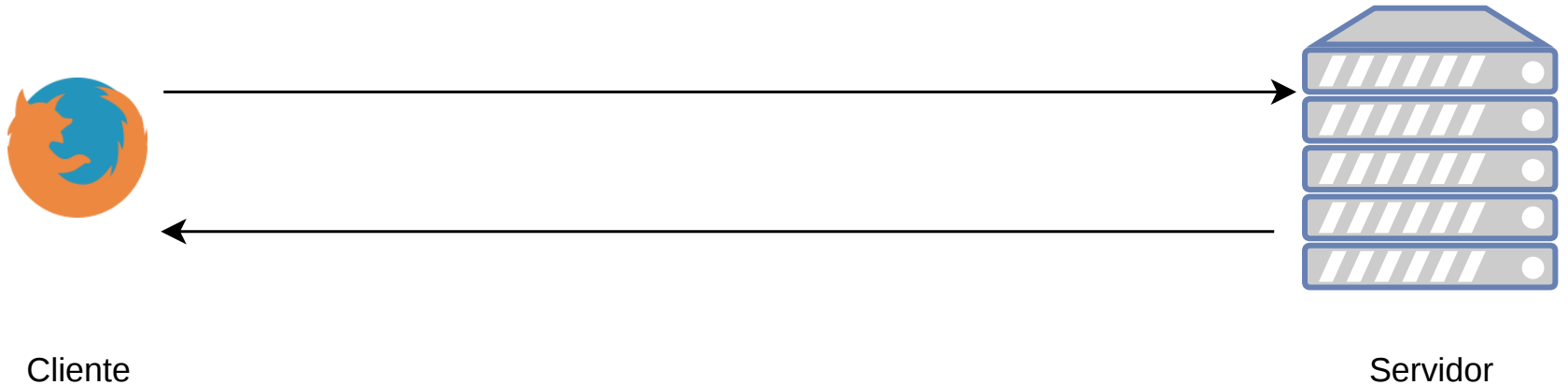
```
String datos = pt.getPeriodictableSoap().getAtomicNumber("Oxygen");

DocumentBuilder db = DocumentBuilderFactory.newInstance()
                                .newDocumentBuilder();
Document doc = db.parse(
    new ByteArrayInputStream(datos.getBytes())
);
String numero = doc.getElementsByTagName("AtomicNumber")
                    .item(0).getTextContent();
System.out.println(numero); // → 8
```

1. INTRODUCCIÓN
2. FORMATOS XML Y JSON
3. SERVICIOS REST
4. PETICIONES AJAX
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. **EL PROTOCOLO HTTPS**
8. BIBLIOGRAFÍA

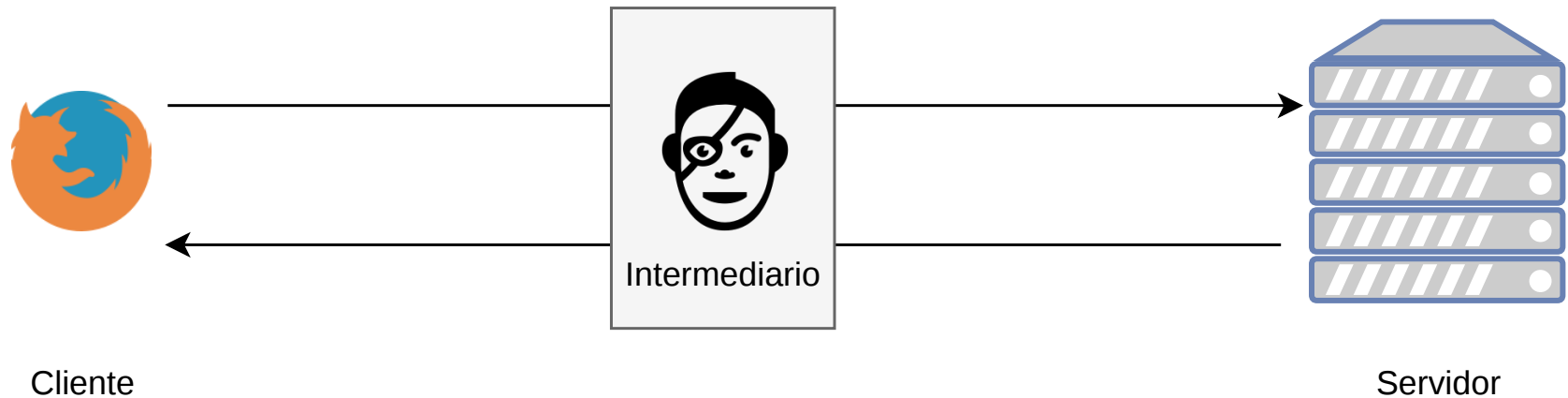


EL PROTOCOLO HTTPS



El protocolo HTTP es un protocolo **inseguro**.

Toda la información que viaja a través de la red (datos de autenticación, formularios, etc.) viaja en forma de texto.



Este esquema es proclive a **ataques de intermediario** (*man in the middle*), en los que un atacante puede interceptar los mensajes que se envían cliente y servidor, pudiendo acceder a su contenido y manipularlos.

Ni el cliente ni el servidor son conscientes de la existencia del intermediario.

El protocolo HTTPS pretende remediar esos problemas añadiendo **encriptación** en los mensajes enviados.

Utiliza técnicas de **criptografía asimétrica**, en los que cada parte utiliza dos claves para el envío de mensajes:

- **Clave privada**, solo conocida por el propietario.
- **Clave publica**, conocida por todos.

Cualquier mensaje encriptado con la clave pública solo puede ser descryptado por la clave privada, y viceversa.

CONTEXTO 1: CONFIDENCIALIDAD DE ENVÍO.

Este esquema de clave pública/privada puede garantizarse la confidencialidad de envío entre dos partes.



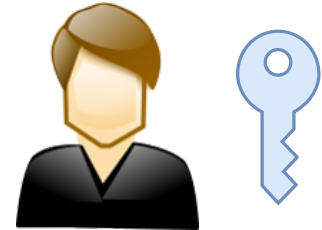
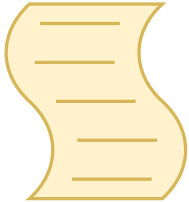
Clave
pública



Clave
privada

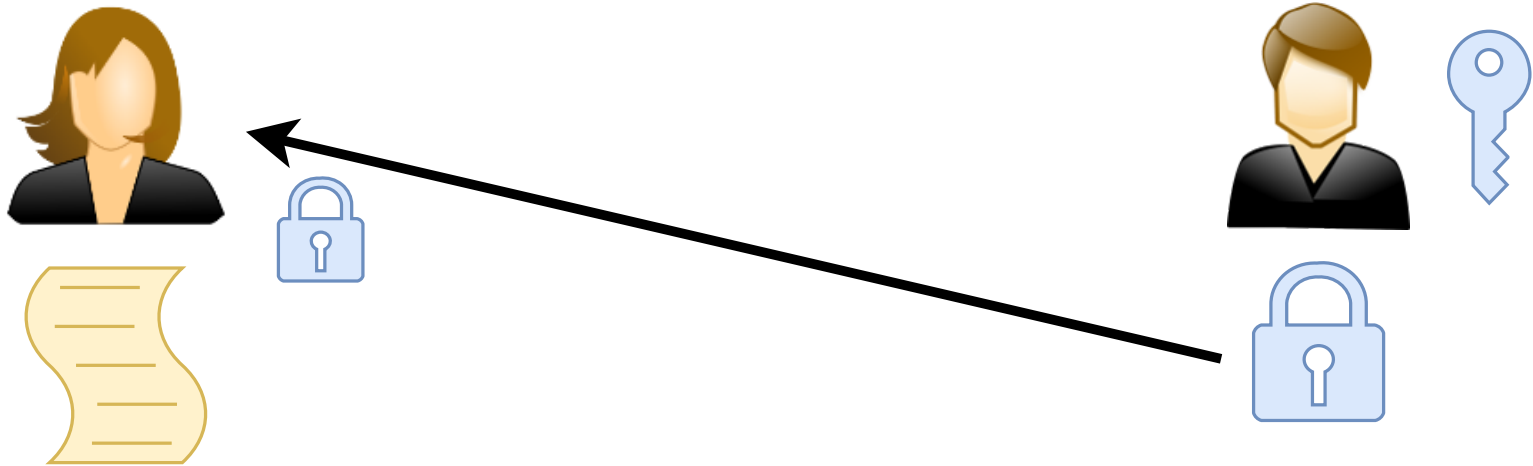
La clave pública tiene el rol de una caja con *candado* y la clave privada tiene el rol de la *llave* que abre el candado.

EJEMPLO



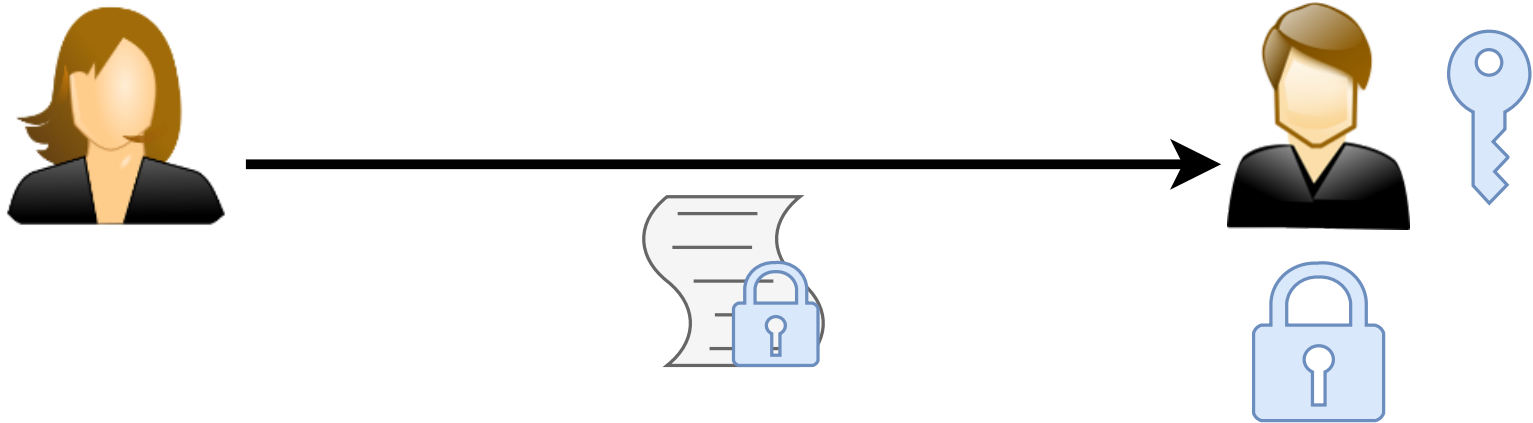
Supongamos que Alice (izqda.) quiere enviar un mensaje a Bob (dcha.)

EJEMPLO



Alice obtiene la clave pública de Bob.

EJEMPLO



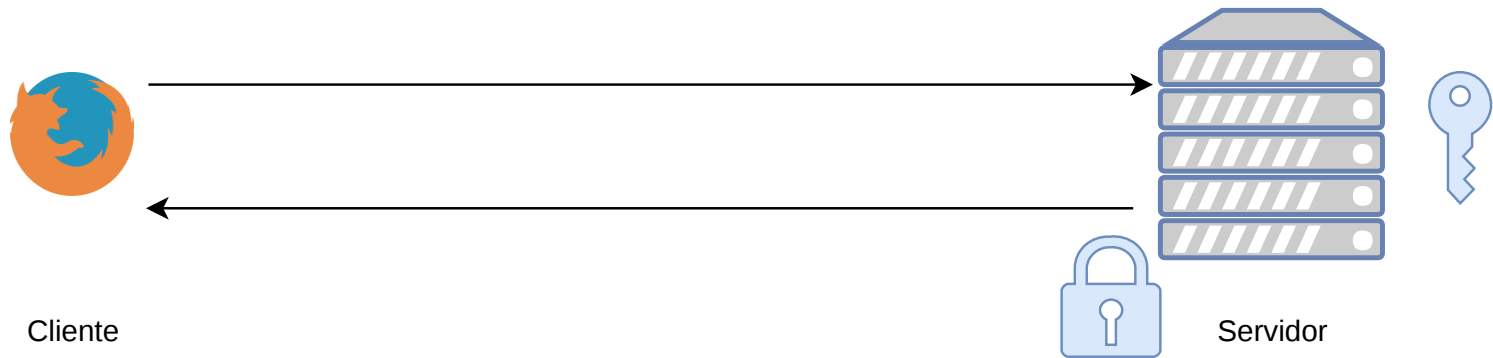
Alice utiliza esta clave pública para encriptar el mensaje.

EJEMPLO



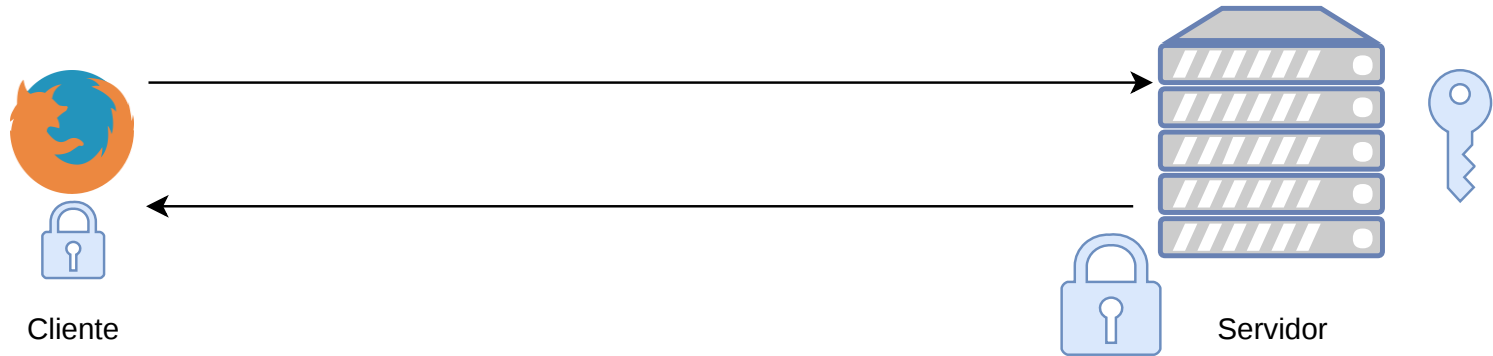
Bob utiliza su clave privada (llave) para desenscriptar el mensaje.

EJEMPLO HTTPS



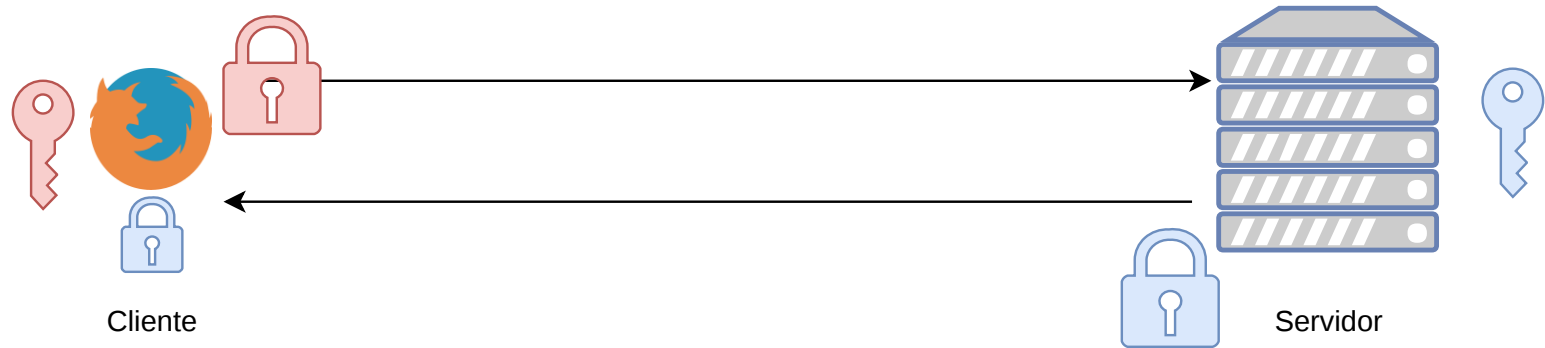
Bajo el protocolo HTTPS, el paso de mensajes funciona de manera similar. El servidor posee dos claves: pública y privada.

EJEMPLO HTTPS



El cliente adquiere la clave pública del servidor.

EJEMPLO HTTPS

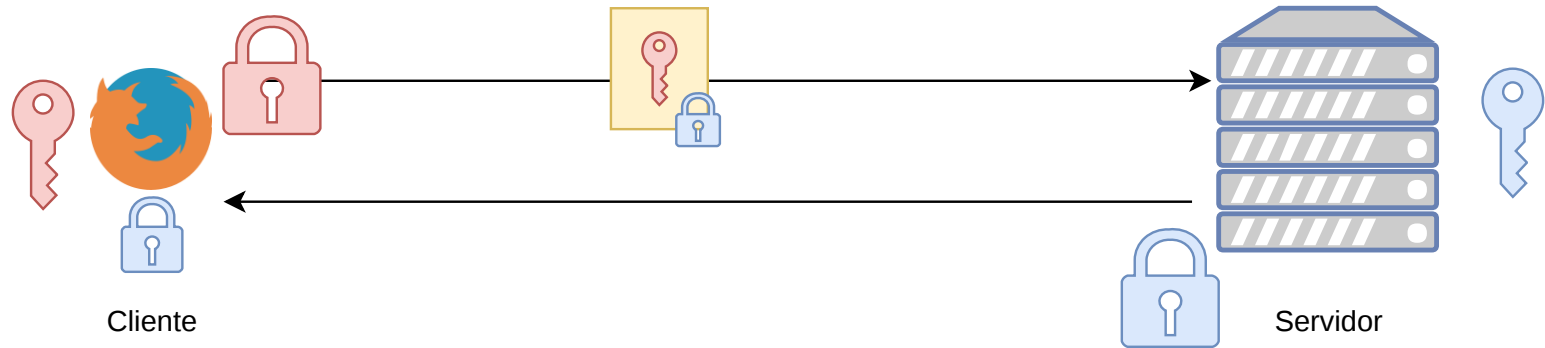


Por otro lado, el cliente genera su propio par de claves pública y privada.

Este par se utiliza solamente en esta petición.

Para la siguiente petición puede generarse un par distinto.

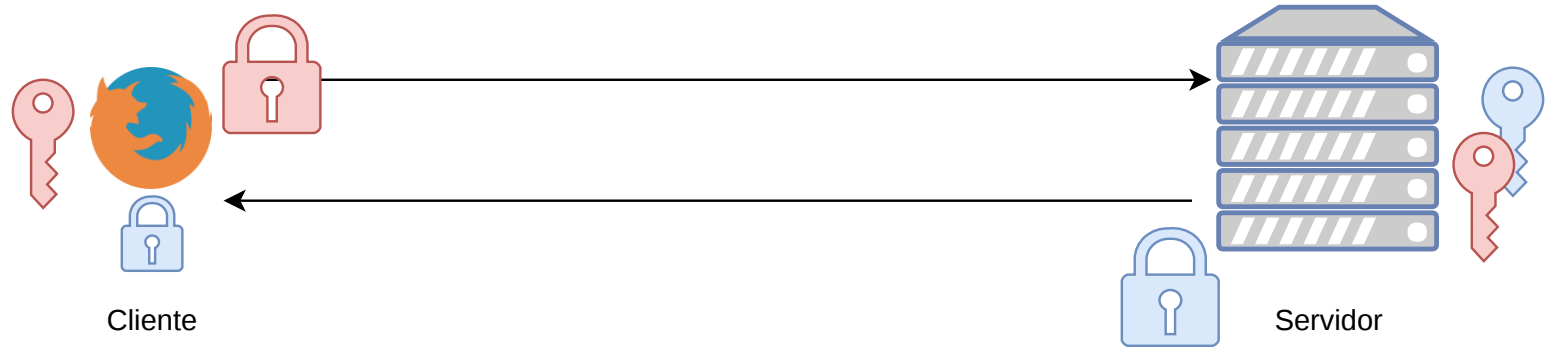
EJEMPLO HTTPS



El cliente envía su clave privada, pero encriptada utilizando la clave pública del servidor.

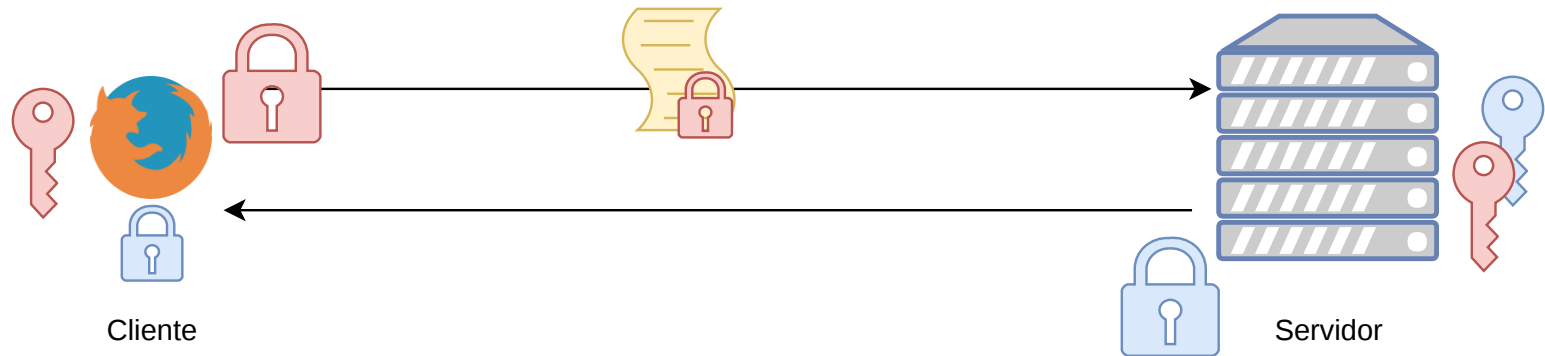
Solo el servidor puede desencriptar el mensaje.

EJEMPLO HTTPS



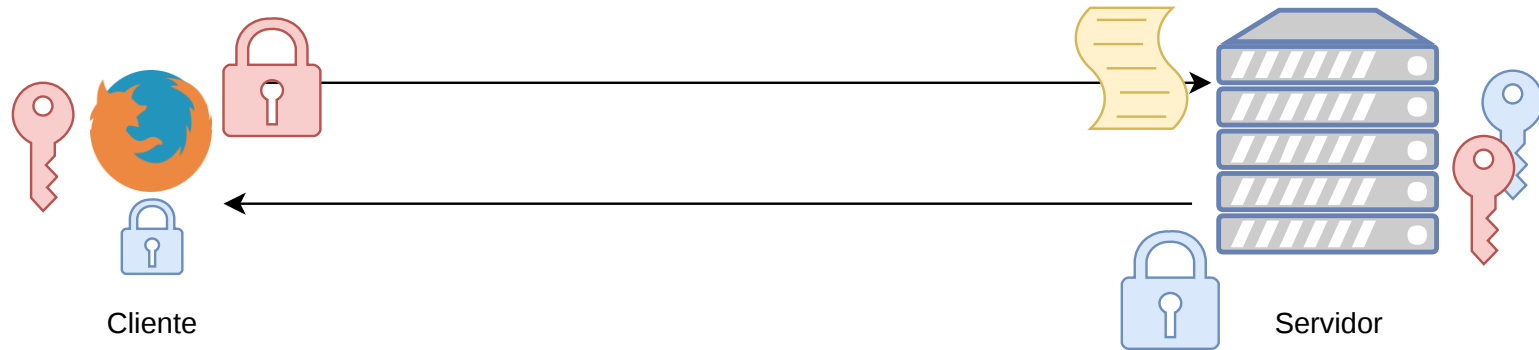
El servidor descripta el mensaje anterior, obteniendo la clave privada que el cliente utilizará en esta petición.

EJEMPLO HTTPS



El cliente pasa a enviar su petición HTTPS encriptada, utilizando su clave pública. Esta petición podrá ser desencryptada por el servidor, ya que este posee la clave privada (roja).

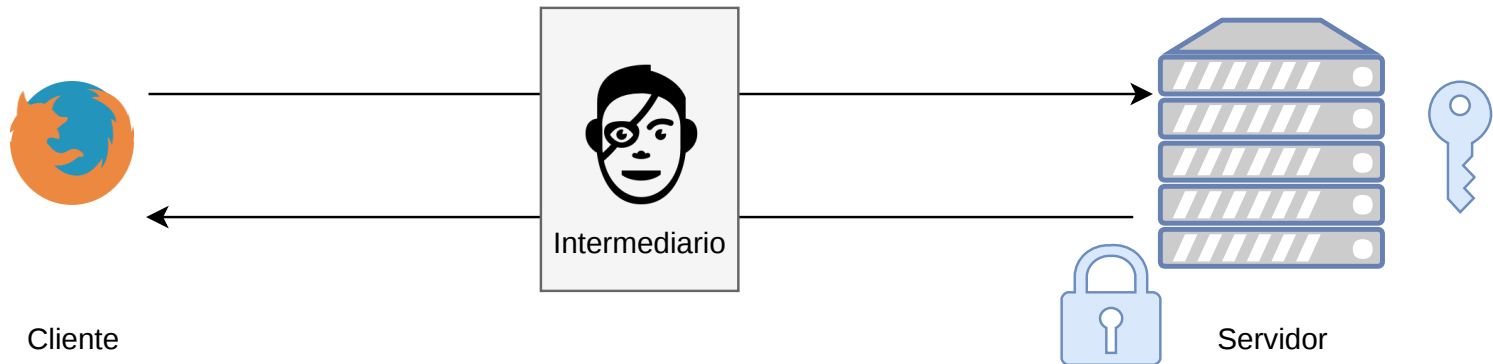
EJEMPLO HTTPS



El servidor descripta la petición.

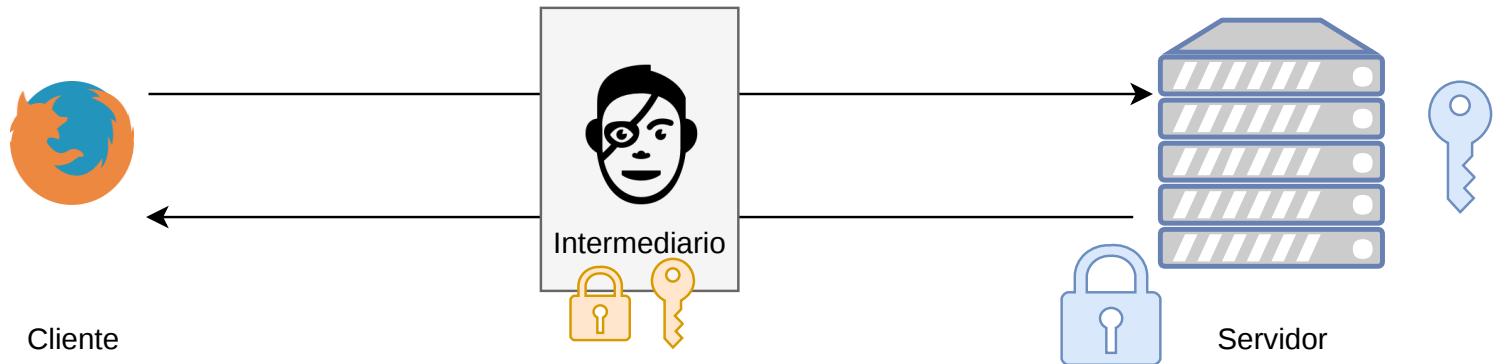
Problema: Este esquema no es inmune a los ataques de intermediario.

ATAQUE DE INTERMEDIARIO



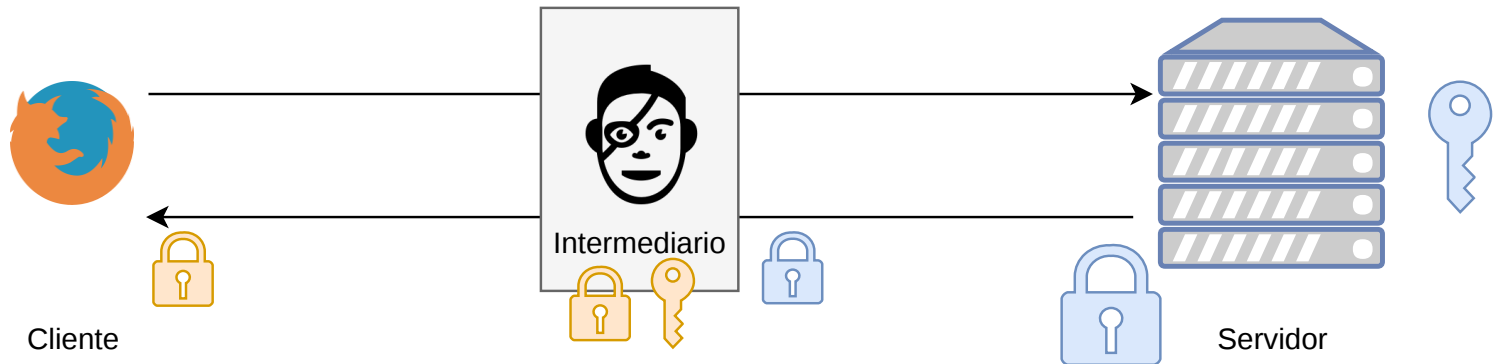
Supongamos que un intermediario malicioso puede acceder y modificar los mensajes que transitan por el canal de comunicación.

ATAQUE DE INTERMEDIARIO



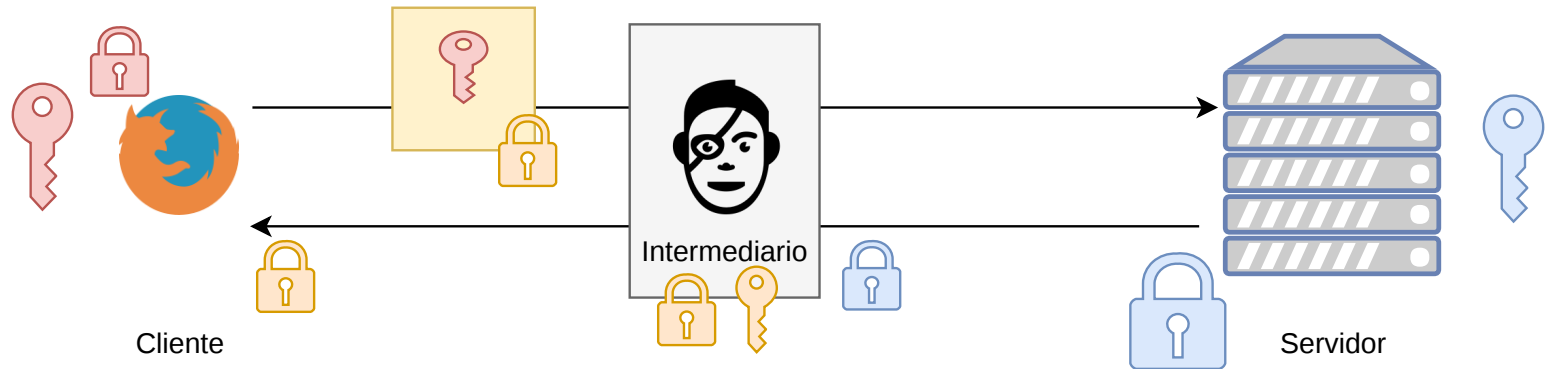
El intermediario genera su propio par de claves publica/privada.

ATAQUE DE INTERMEDIARIO



Cuando el cliente solicita la clave pública del servidor, el intermediario la intercepta y envía al cliente su propia clave pública (amarilla).

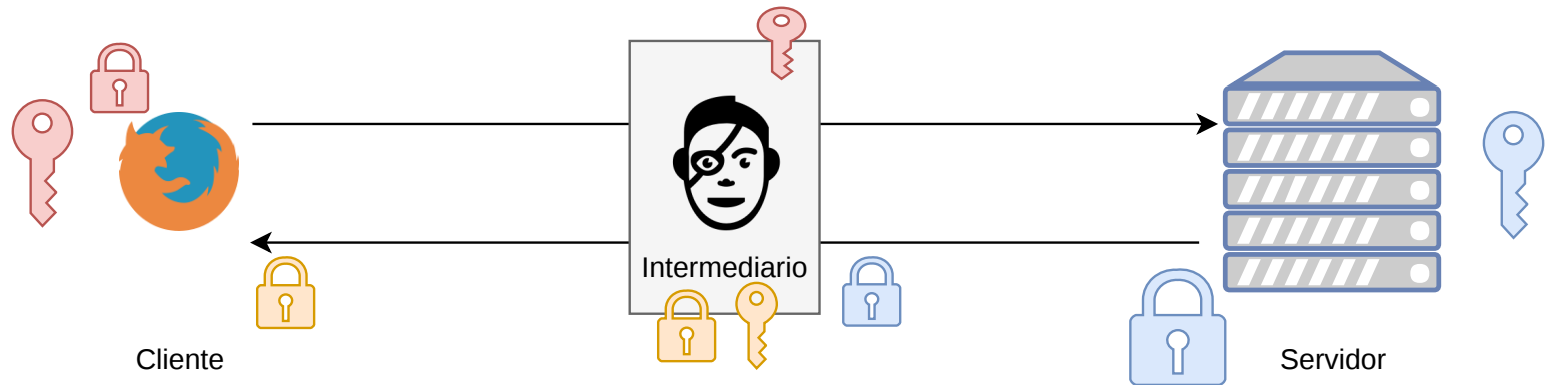
ATAQUE DE INTERMEDIARIO



El cliente, tras generar su propio par de claves pública/privada, envía su clave privada al servidor, encriptada con la clave pública del intermediario.

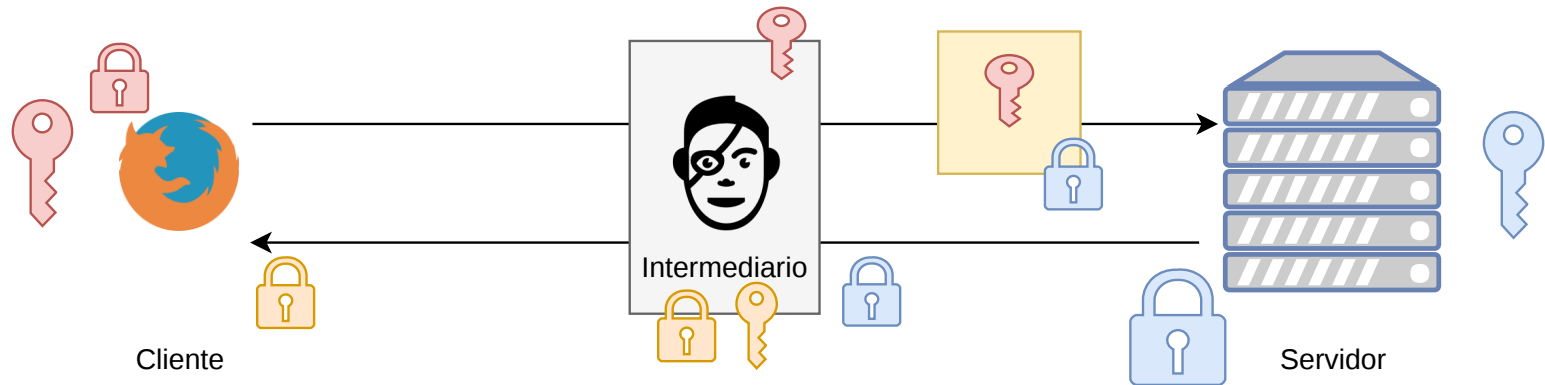
El cliente piensa que la clave pública que posee es la del servidor.

ATAQUE DE INTERMEDIARIO



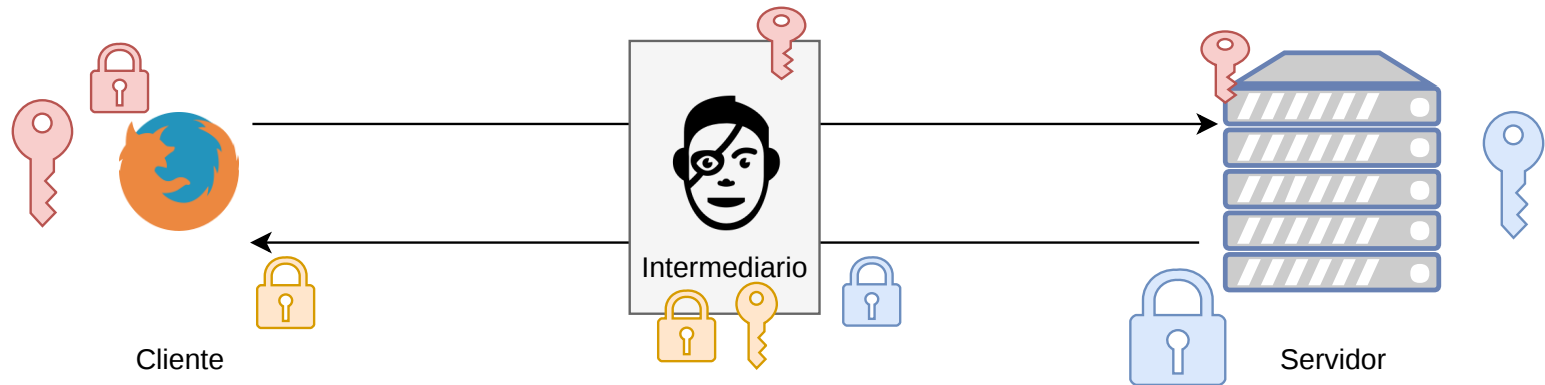
El intermediario descripta el mensaje, obteniendo así la clave privada generada por el cliente.

ATAQUE DE INTERMEDIARIO



Además, el intermediario vuelve a encriptar el mensaje, pero esta vez con la clave pública del servidor, y reenvía el resultado a este último.

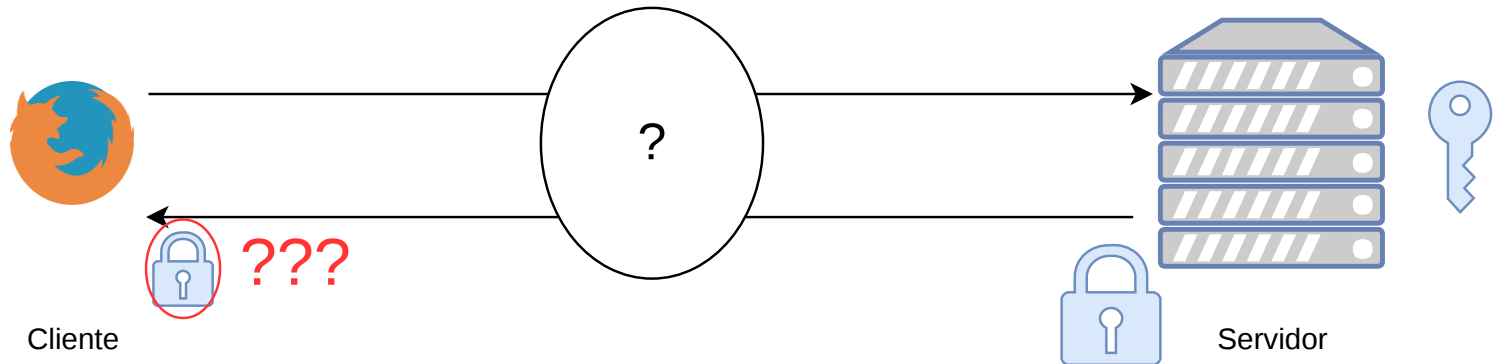
ATAQUE DE INTERMEDIARIO



El servidor descripta el mensaje anterior, obteniendo la clave privada del cliente.

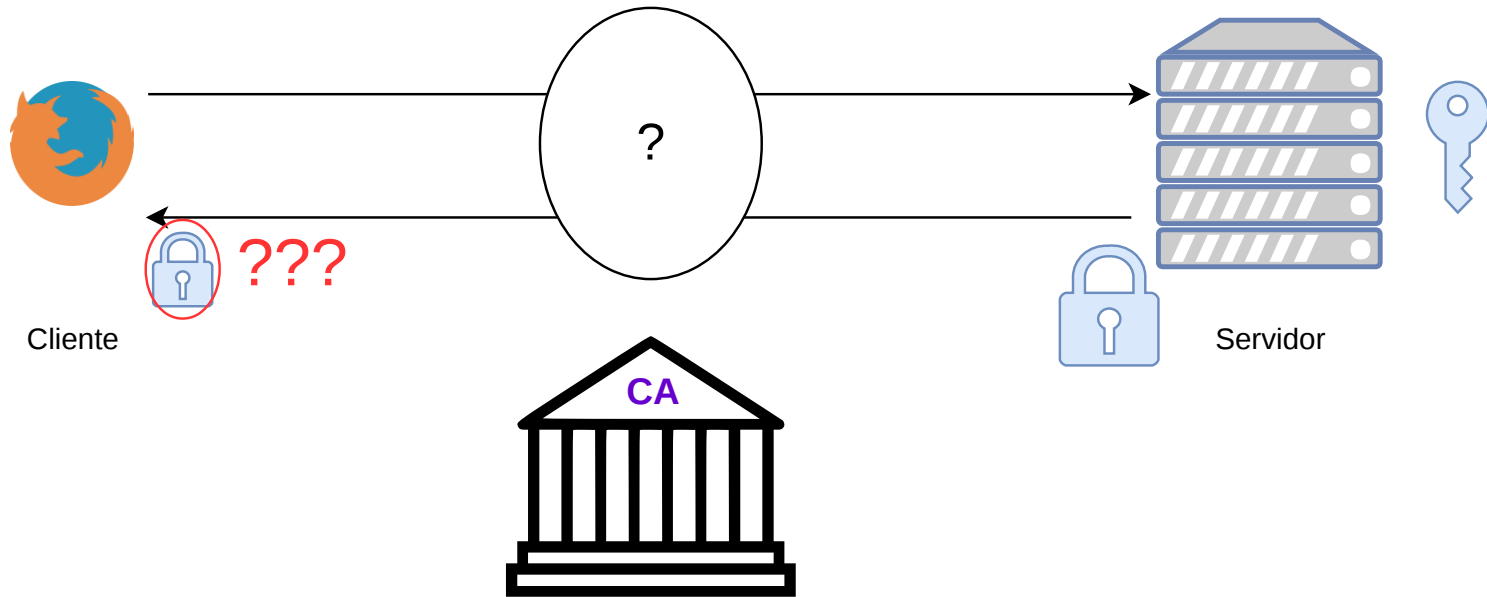
Ninguno de los dos extremos es consciente de que un intermediario ha manipulado los mensajes.

ATAQUE DE INTERMEDIARIO



El problema de fondo es que, cuando el cliente recibe la clave pública del servidor, no hay ninguna manera de comprobar la **autenticidad de esta clave**.

ATAQUE DE INTERMEDIARIO



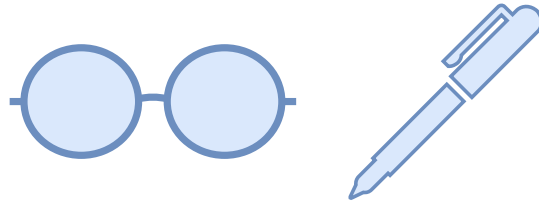
... a menos que el cliente confíe en una **autoridad externa** que le asegure que dicha clave pública proviene del servidor.

Esta autoridad externa recibe el nombre de **Autoridad de Certificación** (Certification Authority).

Esta autoridad puede **firmar** una clave pública si tiene garantías de la procedencia de la misma.

CONTEXTO 2: IDENTIFICACIÓN Y AUTENTIFICACIÓN.

La criptografía de claves públicas y privadas permite garantizar la autenticidad del remitente de los mensajes.

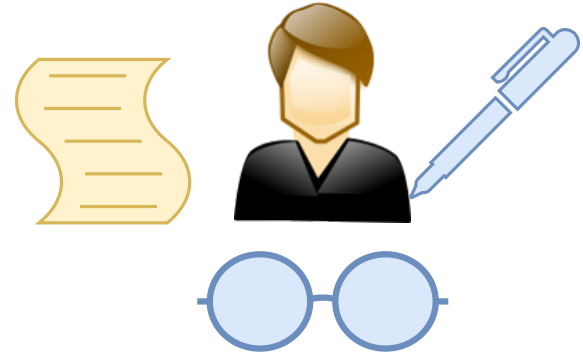


Clave
pública

Clave
privada

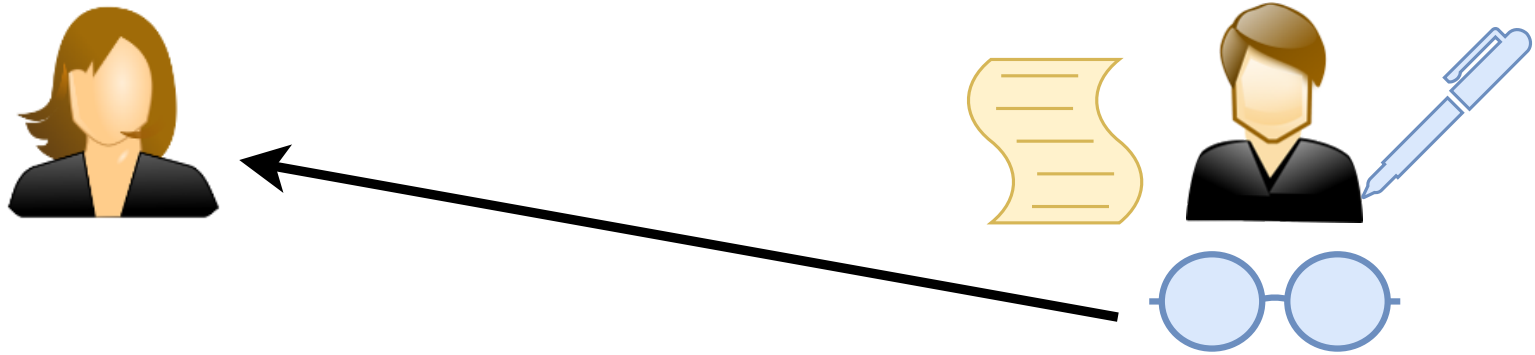
La clave privada tiene el rol de un *bolígrafo* que puede firmar, y la clave pública simula unas *gafas* que permiten comprobar la autenticidad de las firmas del bolígrafo.

EJEMPLO



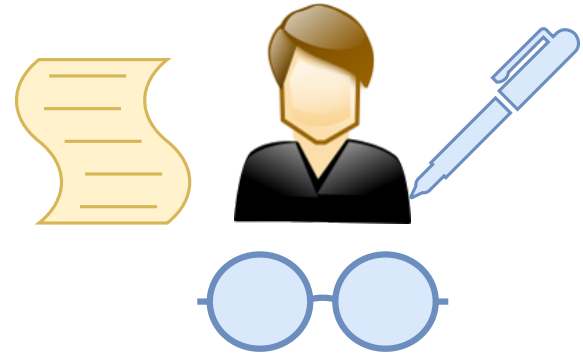
Supongamos que Alice (izqda.) quiere recibir un mensaje de Bob (dcha.), con la garantía de que el remitente del mensaje es, realmente, Bob.

EJEMPLO



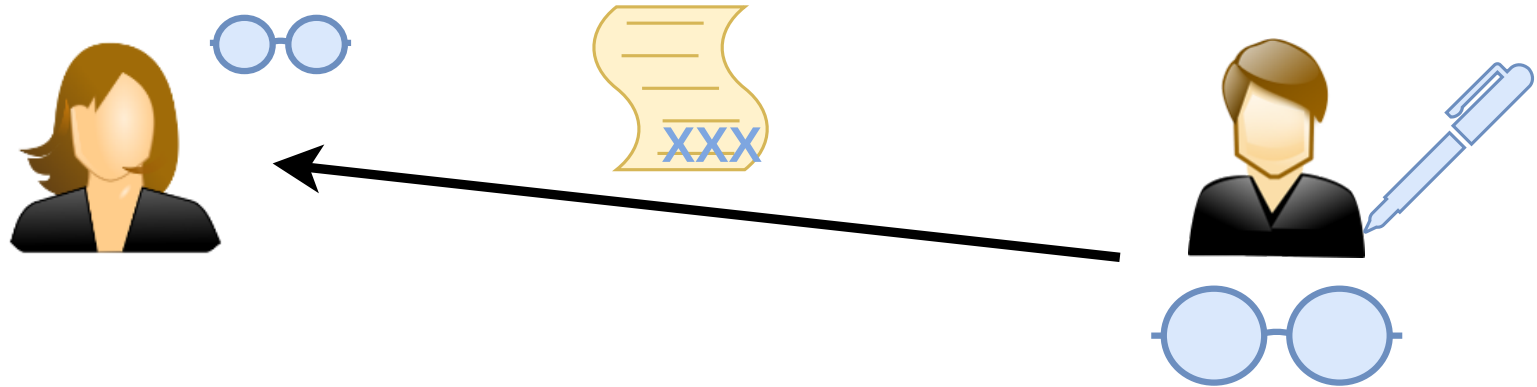
Alice obtiene la clave pública de Bob.

EJEMPLO



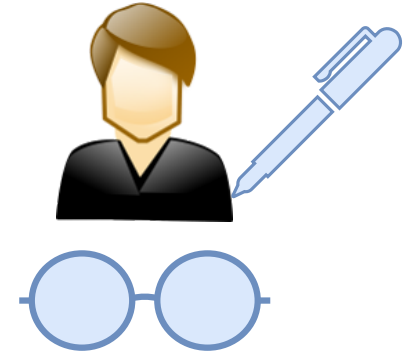
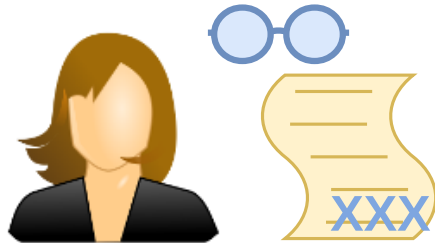
Alice obtiene la clave pública de Bob.

EJEMPLO



Bob **firma** (encripta) el mensaje con su clave privada, y lo envía a Alice.

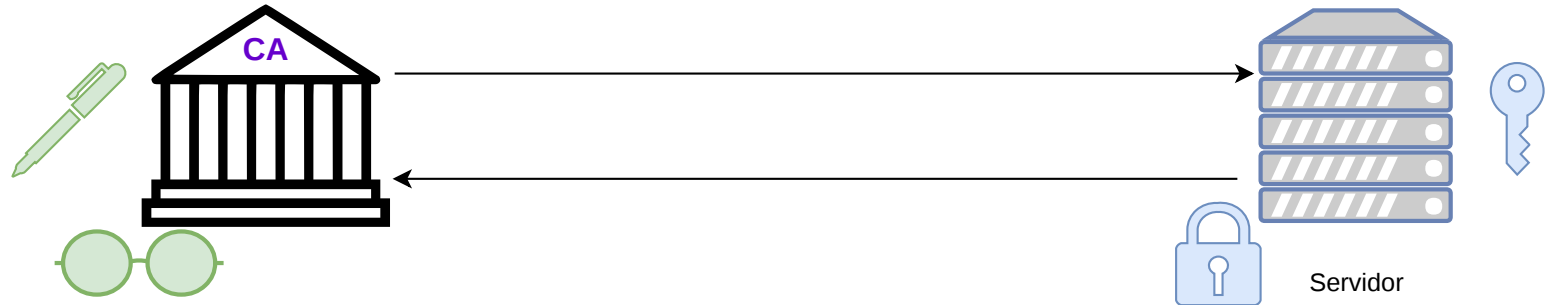
EJEMPLO



Alice recibe el mensaje encriptado y lo descripta utilizando la clave pública. Esta clave solo puede descriptar mensajes cifrados con la clave privada correspondiente, por lo que el remitente ha de ser Bob.

Solo Bob puede encriptar los mensajes descifrables con la su clave pública.

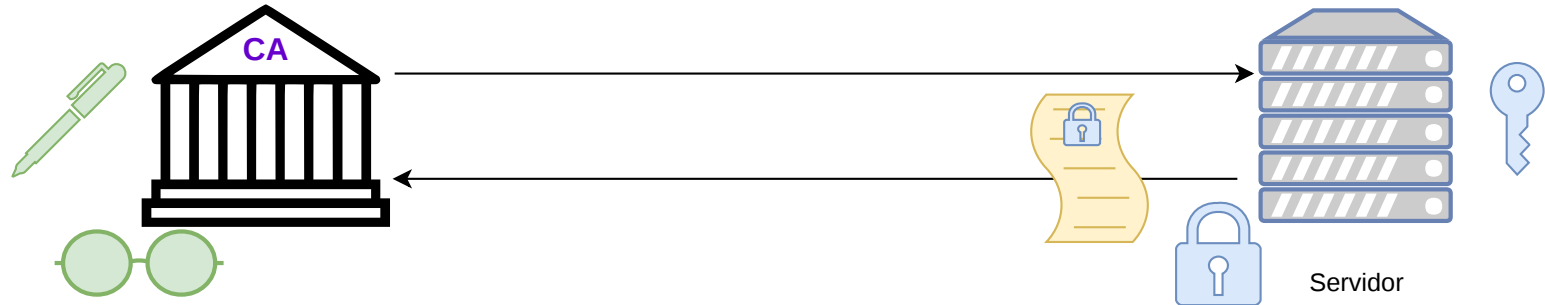
AUTORIDAD DE CERTIFICACIÓN



Volvemos al contexto de cliente-servidor con autoridad certificadora autorizadora.

El servidor quiere que la CA certifique la autenticidad de su clave privada.

AUTORIDAD DE CERTIFICACIÓN

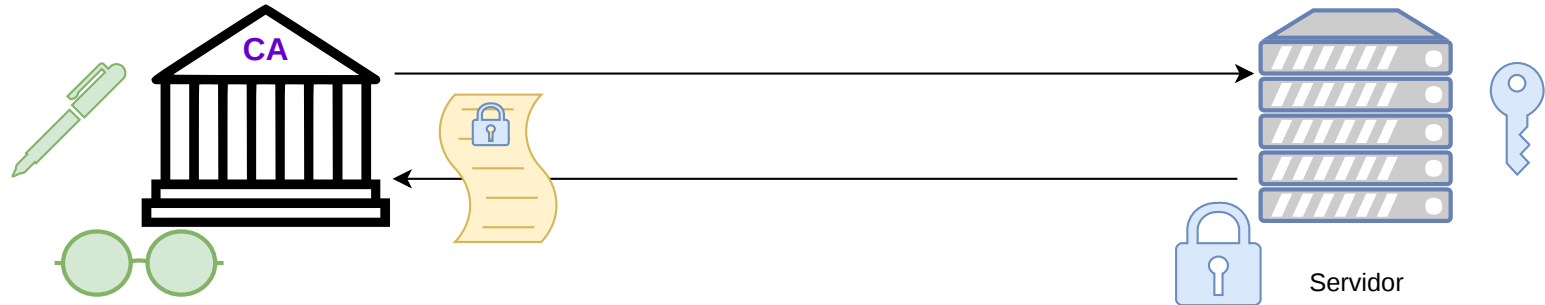


El servidor genera un **certificado**, que contiene:

- Datos del servidor: nombre, dominio, organización, etc.
- Clave pública del servidor.

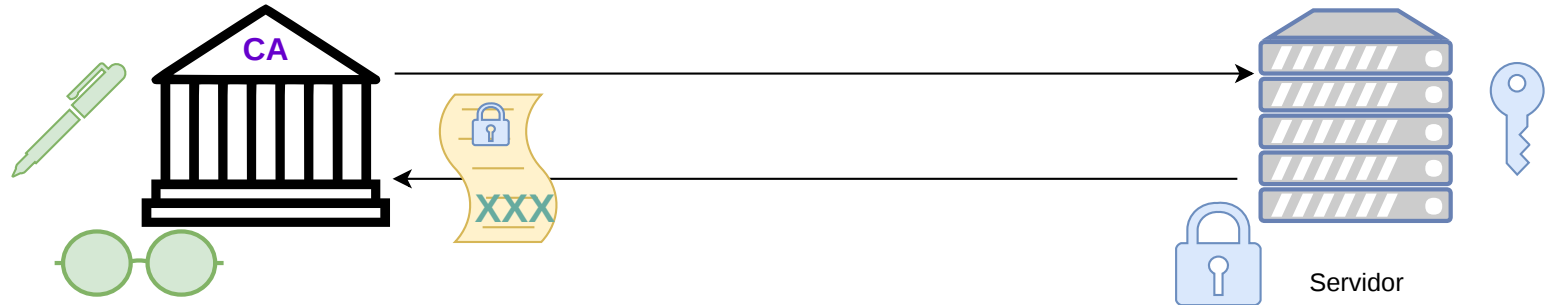
Este certificado está sin firmar.

AUTORIDAD DE CERTIFICACIÓN



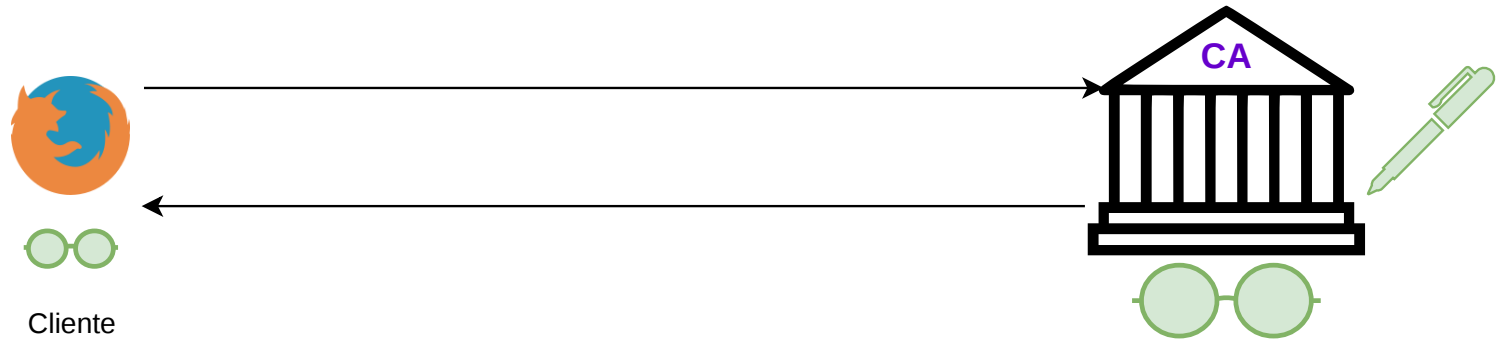
El servidor envía el certificado sin firmar a la CA para que esta última lo firme.

AUTORIDAD DE CERTIFICACIÓN



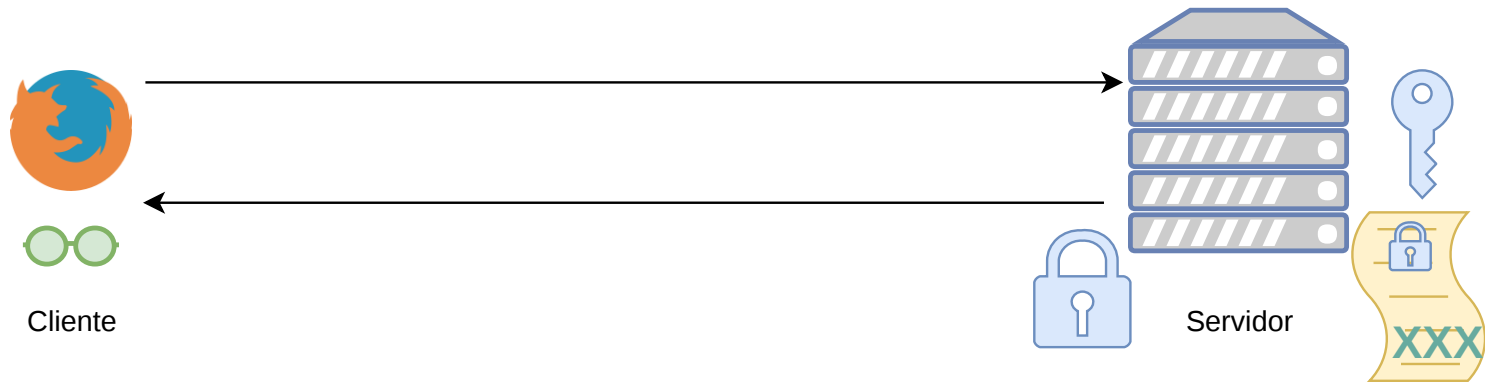
La CA, tras comprobar (posiblemente por medios no telemáticos) que los datos del certificado concuerdan con la identidad de quién solicita su firma, procede a firmar el certificado.

AUTORIDAD DE CERTIFICACIÓN



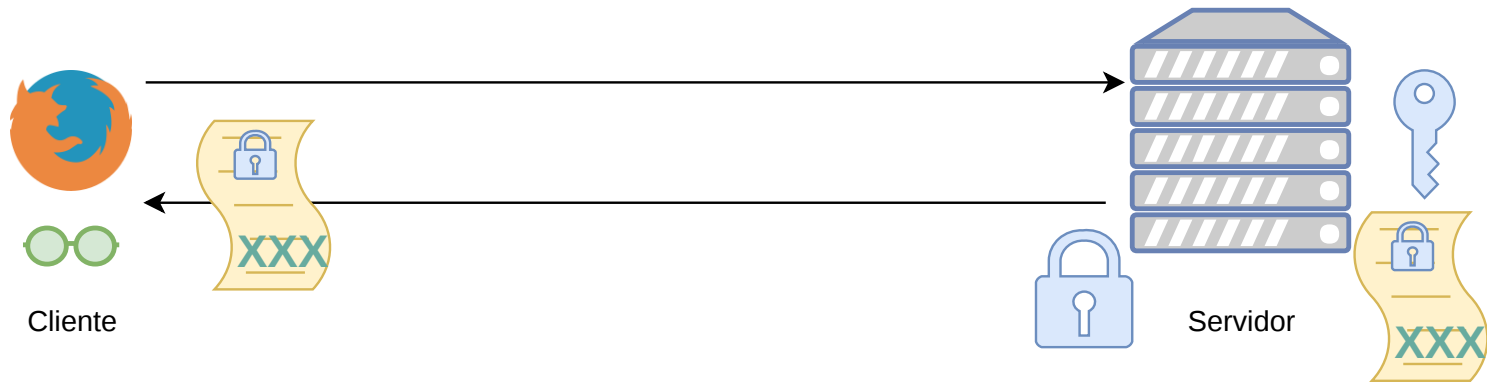
Por otro lado, suponemos que el navegador **conoce** a la entidad certificadora, y que este viene instalado con la clave pública de la CA, contenida dentro de un **certificado raíz** (CA *root certificate*)

COMUNICACIÓN CERTIFICADA CLIENTE/SERVIDOR



Cuando el cliente solicita la clave pública al servidor, este último le envía la clave pública metida dentro del certificado con la firma de la CA.

COMUNICACIÓN CERTIFICADA CLIENTE/SERVIDOR



El cliente recibe este certificado y comprueba (utilizando la clave pública de la CA), que la firma de la CA es auténtica.

Como el cliente confía en la CA, tiene la certeza de que la clave pública es del servidor cuyos datos constan en el certificado.

GENERACIÓN DE CLAVES, FIRMAS Y CERTIFICADOS

El protocolo HTTPS utiliza los protocolos criptográficos TLS/SSL como mecanismo en la capa de transporte.

OpenSSL es un conjunto de herramientas relacionadas con estos dos protocolos.

<https://www.openssl.org/>

Algunas de estas herramientas sirven para crear claves, certificados y firmas.

GENERAR CLAVES PÚBLICA Y PRIVADA

```
# openssl genrsa -out mi_clave.pem 1024
```

Se genera un fichero **mi_clave.pem** con un par de claves publica/privada.



mi_clave.pem

GENERAR CERTIFICADO SIN FIRMAR

```
# openssl req -new -key mi_clave.pem -out certificado.csr
```

Tras solicitar información que se adjuntará en el certificado (host, organización, ciudad, etc.), se genera un fichero **certificado.csr** con la clave pública.

El certificado no tiene firma.



certificado.csr

FIRMAR CERTIFICADO

El certificado generado deberá enviarse a una autoridad de certificación para que sea firmado por ésta.

Cuando la CA compruebe que el remitente del certificado es quién dice ser, firmará dicho certificado con su clave privada:

```
# openssl x509 -req -in certificado.csr -signkey clave_ca.key  
-out certificado_firmado.crt
```

donde *clave_ca.key* es la clave privada de CA que firma el certificado.

CERTIFICADOS AUTOFIRMADOS

En entornos de desarrollo puede no ser posible o viable el acceso a una CA que firme nuestros certificados.

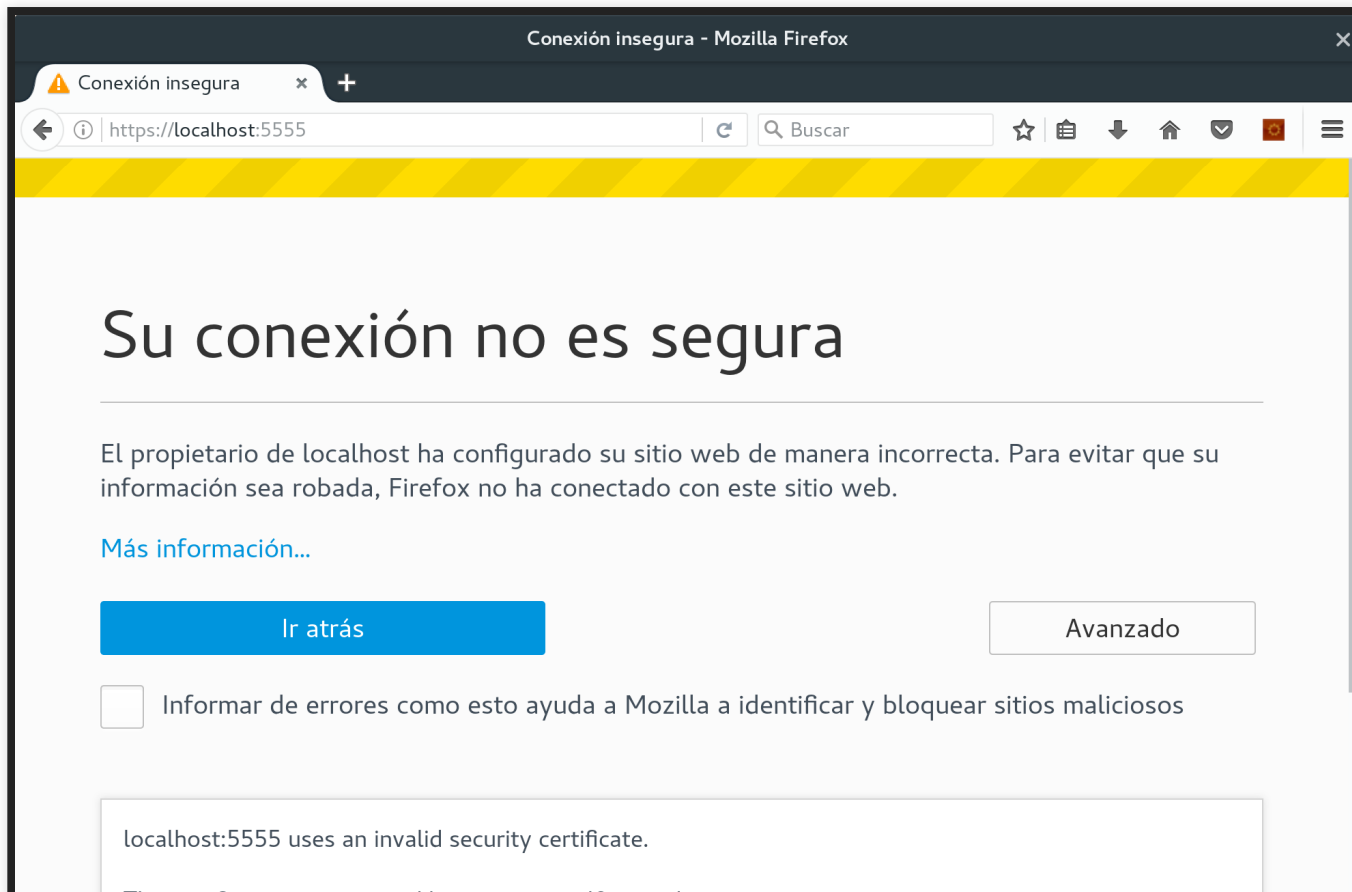
Como alternativa, los certificados pueden ser firmados por la misma entidad que los genera:

```
# openssl x509 -req -in certificado.csr -signkey mi_clave.pem  
-out certificado_firmado.crt
```



certificado_firmado.crt

No obstante, los navegadores web no confían (obviamente) en esta clase de certificados, por lo que muestran un mensaje de advertencia al intentar establecer una conexión HTTPS con este tipo de certificados.



CREAR UN SERVIDOR HTTPS EN NODE

Al igual que el módulo `http` (ver Tema 4), existe un módulo core `https` para crear servidores web HTTPS.

```
var https = require("https");
var fs = require("fs");

var clavePrivada = fs.readFileSync("./miclave.key");
var certificado = fs.readFileSync("./certificado_firmado.crt");

var servidor = https.createServer(
  { key: clavePrivada, cert: certificado },
  function(request, response) {
    // Cada vez que se reciba una petición
    // se llamará a esta función callback
  });
```

El puerto por defecto para el protocolo HTTPS es el 443.
No obstante, en desarrollo suele utilizarse un puerto distinto
(por ejemplo, el 5555).

```
servidor.listen(5555, function(err) {  
  console.log("Escuchando en puerto 5555");  
});
```

INTEGRAR https CON EXPRESS.JS

A la hora de crear un servidor, puede utilizarse una aplicación Express.js como callback de peticiones.

```
var express = require("express");
var https = require("https");
var fs = require("fs");
// ...
var app = express();

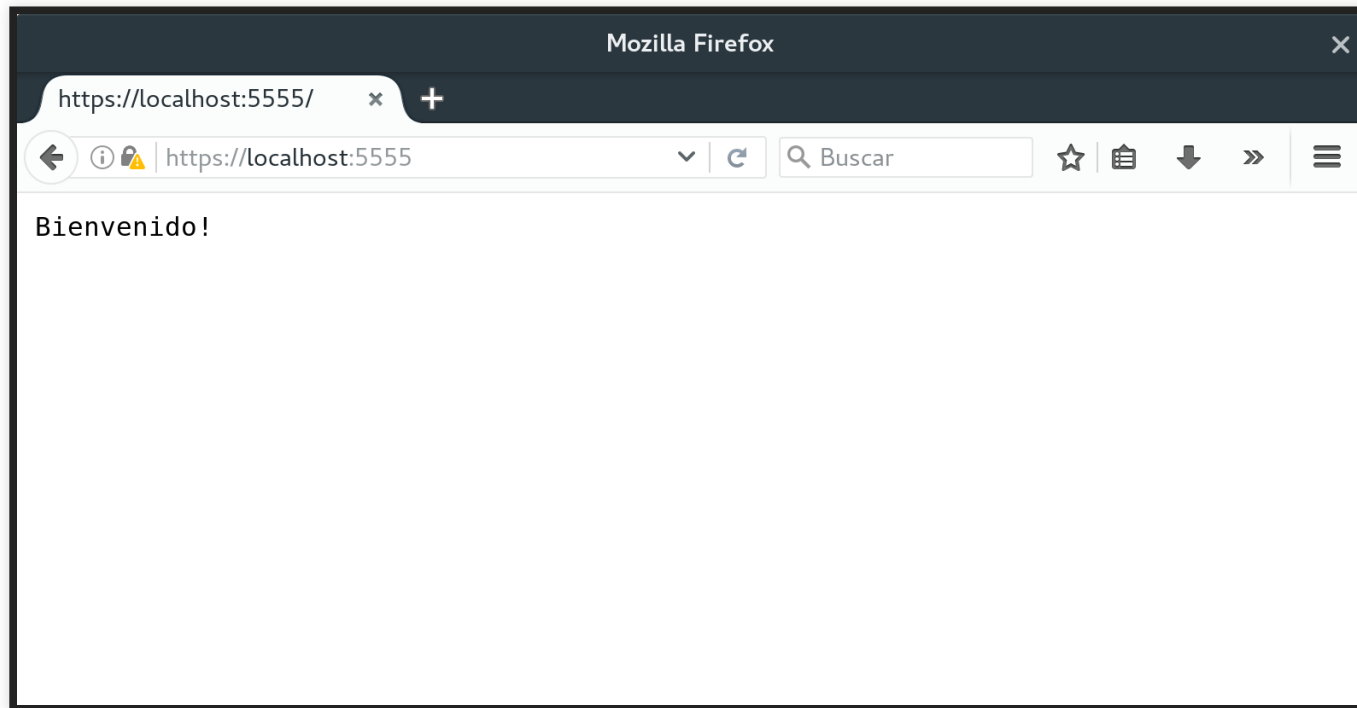
app.get("/", function(request, response) {
  response.end("Bienvenido!");
});

var servidor = https.createServer(
  { key: clavePrivada, cert: certificado }, app);

servidor.listen(5555, function(err) { ... });
```


Al acceder a <https://localhost:5555/> en el navegador se nos mostrará un mensaje advirtiéndonos de que el certificado está autofirmado.

Si el usuario acepta el certificado, se accede al recurso solicitado:



1. INTRODUCCIÓN
2. FORMATOS XML Y JSON
3. SERVICIOS REST
4. PETICIONES AJAX
5. AUTENTICACIÓN BÁSICA
6. INTRODUCCIÓN A SOAP
7. EL PROTOCOLO HTTPS
8. **BIBLIOGRAFÍA**



BIBLIOGRAFÍA

- V. Bojinov
RESTful Web API Design with Node.js,
2nd edition
Packt Publishing, 2016
- B. Bibeault, Y. Katz, A. De Rosa
jQuery in Action, 3rd edition
Manning Publications, 2015

