

# Compte-rendu tp 4

## les explications nécessaires pour comprendre démarche

### étape 1

Déterminer maintenant le rôle de la fonction `after()` pour la déclaration de la route `'/secu'` :

- utilisée pour spécifier des opérations supplémentaires qui doivent être effectuées après que la route principale a été configurée.
- inclure l'ajout de gestionnaires de validation, l'application de logique spécifique, ou toute autre tâche qui doit être exécutée après la configuration initiale de la route.

### étape 2

1. Créer une nouvelle clé RSA de 2048 bits appelé `[server.key]`.
  - `openssl genrsa -out server.key 2048`

2. Créer un fichier de Certificate Signing Request `[server.req]`
  - `openssl req -new -key server.key -out server.req`

```
C:\Users\xingt\PhppstormProjects\NodeJS_tp\Secu1-tp4>openssl req -new -key server.key -out server.req
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. effectuer la signature avec la clé privée
  - `openssl x509 -req -days 365 -in server.req -signkey server.key -out server.crt`

```
C:\Users\xingt\PhppstormProjects\NodeJS_tp\Secu1-tp4>openssl x509 -req -days 365 -in server.req -signkey server.key -out server.crt
Certificate request self-signature ok
subject=C=AU, ST=Some-State, O=Internet Widgits Pty Ltd
```

4. tester le certificat généré
  - `openssl s_server -accept 4567 -cert server.crt -key server.key -www -state`
  - `--https://localhost:4567/` dans Postman

Commenter l'affichage retourné dans Postman :

```
4 s_server -accept 4567 -cert server.crt -key server.key -www -state
```

- `s_server`: Commande pour lancer un serveur TLS/SSL.
- `-accept 4567`: Spécifie le port d'écoute du serveur (4567).
- `-cert server.crt`: Spécifie le fichier du certificat du serveur.
- `-key server.key`: Spécifie le fichier de la clé privée du serveur.
- `-www`: Active le mode web pour afficher les informations dans le navigateur.
- `-state`: Active l'enregistrement de l'état.

```

6  Ciphers supported in s_server binary
7  TLSv1.3      :TLS_AES_256_GCM_SHA384      TLSv1.3      :TLS_CHACHA20_POLY1305_SHA256
8  TLSv1.3      :TLS_AES_128_GCM_SHA256      TLSv1.2      :ECDHE-ECDSA-AES256-GCM-SHA384
9  TLSv1.2      :ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2      :DHE-RSA-AES256-GCM-SHA384
10 TLSv1.2      :ECDHE-ECDSA-CHACHA20-POLY1305 TLSv1.2      :ECDHE-RSA-CHACHA20-POLY1305
11 TLSv1.2      :DHE-RSA-CHACHA20-POLY1305 TLSv1.2      :ECDHE-ECDSA-AES128-GCM-SHA256
12 TLSv1.2      :ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2      :DHE-RSA-AES128-GCM-SHA256
13 TLSv1.2      :ECDHE-ECDSA-AES256-SHA384 TLSv1.2      :ECDHE-RSA-AES256-SHA384
14 TLSv1.2      :DHE-RSA-AES256-SHA256      TLSv1.2      :ECDHE-ECDSA-AES128-SHA256
15 TLSv1.2      :ECDHE-RSA-AES128-SHA256 TLSv1.2      :DHE-RSA-AES128-SHA256
16 TLSv1.0      :ECDHE-ECDSA-AES256-SHA      TLSv1.0      :ECDHE-RSA-AES256-SHA
17 SSLv3        :DHE-RSA-AES256-SHA      TLSv1.0      :ECDHE-ECDSA-AES128-SHA
18 TLSv1.0      :ECDHE-RSA-AES128-SHA      SSLv3        :DHE-RSA-AES128-SHA
19 TLSv1.2      :RSA-PSK-AES256-GCM-SHA384 TLSv1.2      :DHE-PSK-AES256-GCM-SHA384
20 TLSv1.2      :RSA-PSK-CHACHA20-POLY1305 TLSv1.2      :DHE-PSK-CHACHA20-POLY1305
21 TLSv1.2      :ECDHE-PSK-CHACHA20-POLY1305 TLSv1.2      :AES256-GCM-SHA384
22 TLSv1.2      :PSK-AES256-GCM-SHA384      TLSv1.2      :PSK-CHACHA20-POLY1305
23 TLSv1.2      :RSA-PSK-AES128-GCM-SHA256 TLSv1.2      :DHE-PSK-AES128-GCM-SHA256
24 TLSv1.2      :AES128-GCM-SHA256          TLSv1.2      :PSK-AES128-GCM-SHA256
25 TLSv1.2      :AES256-SHA256             TLSv1.2      :AES128-SHA256

```

- Liste des Ciphers (algorithmes de chiffrement) supportés pour TLSv1.3 et TLSv1.2

```

38  Ciphers common between both SSL end points:
39  TLS_AES_128_GCM_SHA256      TLS_AES_256_GCM_SHA384      TLS_CHACHA20_POLY1305_SHA256
40  ECDHE-RSA-AES128-GCM-SHA256 ECDHE-ECDSA-AES128-GCM-SHA256 ECDHE-RSA-AES256-GCM-SHA384
41  ECDHE-ECDSA-AES256-GCM-SHA384 ECDHE-ECDSA-CHACHA20-POLY1305 ECDHE-RSA-CHACHA20-POLY1305
42  ECDHE-ECDSA-AES128-SHA      ECDHE-RSA-AES128-SHA      ECDHE-ECDSA-AES256-SHA
43  ECDHE-RSA-AES256-SHA      AES128-GCM-SHA256      AES256-GCM-SHA384
44  AES128-SHA      AES256-SHA

```

- Ciphers communs entre serveur et client

```

45  Signature Algorithms: ECDSA+SHA256:RSA-PSS+SHA256:RSA+SHA256:ECDSA+SHA384:RSA-PSS+SHA384:RSA
    +SHA384:RSA-PSS+SHA512:RSA+SHA512:RSA+SHA1
46  Shared Signature Algorithms: ECDSA+SHA256:RSA-PSS+SHA256:RSA+SHA256:ECDSA+SHA384:RSA-PSS+SHA384:RSA
    +SHA384:RSA-PSS+SHA512:RSA+SHA512
47  Supported groups: x25519:secp256r1:secp384r1
48  Shared groups: x25519:secp256r1:secp384r1

```

- Algorithme de signature et groupes de chiffrement supportés

New, TLSv1.3, Cipher is TLS\_AES\_128\_GCM\_SHA256

SSL-Session:

```
Protocol   : TLSv1.3
Cipher     : TLS_AES_128_GCM_SHA256
Session-ID: 86A037EC576C729C7A20BE4DB3F018027CEF5CFCA596468CF60C56E2D7EC5838
Session-ID-ctx: 01000000
Resumption PSK: 73F74D39C48EB2BF77668BA924CB0952B4BA4AAB009964BB006FC79B3952E86A
PSK identity: None
PSK identity hint: None
SRP username: None
Start Time: 1708353937
Timeout    : 7200 (sec)
Verify return code: 0 (ok)
Extended master secret: no
Max Early Data: 0
```

#### - Informations de session SSL

Protocole TLSv1.3, Cipher utilisé (TLS\_AES\_128\_GCM\_SHA256), ID de session ...

```
66 | 0 items in the session cache
67 | 0 client connects (SSL_connect())
68 | 0 client renegotiates (SSL_connect())
69 | 0 client connects that finished
70 | 4 server accepts (SSL_accept())
71 | 0 server renegotiates (SSL_accept())
72 | 1 server accepts that finished
73 | 0 session cache hits
74 | 0 session cache misses
75 | 0 session cache timeouts
76 | 0 callback cache hits
77 | 0 cache full overflows (128 allowed)
```

#### - Statistiques de session SSL

Détails sur les connexions client et serveur, les acceptations ...

79 no client certificate available

#### - Indique qu'aucun certificat client n'a été fourni

**readFileSync() utilise une fonction synchrone. Vous réfléchirez à l'impact de cet usage sur le serveur, et adapter si nécessaire pour une meilleure optimisation :**

- peut bloquer le thread principal du serveur pendant le temps de lecture du fichier.
- problèmes de performance lorsque génération de plusieurs requêtes simultanées

- version asynchrone ⇒ readFile, effectuer la lecture de manière asynchrone.

- permet au serveur de continuer à traiter d'autres requêtes pendant que le fichier est en cours de lecture

HTTPS (Hyper Text Transfer Protocol Secure)

### étape 3

1. générer la clé privée ECDSA :
  - openssl ecparam -genkey -name prime256v1 -noout -out ec\_private.pem
2. extraire la clé publique à partir de la clé privée :
  - openssl ec -in ec\_private.pem -pubout -out ec\_public.pem

clé privée qui est utilisée pour signer le JWT ⇒ connexion, authentification

clé publique pour vérifier les JWT entrants ⇒ données

### vos choix, ce qui a marché facilement

#### étape 3

Pour vérifier qu'on a 1 token avec un bon payload, on utilise : <https://jwt.io/>.

Au début, j'ai choisi jsonwebtoken, qui marche parfaitement. Mais je trouve que c'est bizarre que les informations qu'on a écrit dans jwt.js ne sont jamais utilisées, donc après avoir lu la [documentation](#), j'ai trouvé le moyen qui écrit moins et fonctionne parfaitement, voici les captures de mes résultats :

1. créer un compte utilisateur n'existe pas

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/signup`. The request body is a JSON object: `{ "email": "x", "password": "qsd" }`. The response status is `200 OK` with a response time of `31 ms` and a body size of `339 B`. The response body is a JSON object: `{ "message": "Utilisateur enregistré avec succès", "newUser": { "email": "x", "password": "0b1b8e9fc13fa4c07a6d7983ee14619bd71f2ea4001c2c6dba9ab79da4920d5c", "role": "admin" } }`.

```
POST http://localhost:3000/signup

{
  "email": "x",
  "password": "qsd"
}
```

200 OK 31 ms 339 B Save as example

```
{
  "message": "Utilisateur enregistré avec succès",
  "newUser": {
    "email": "x",
    "password": "0b1b8e9fc13fa4c07a6d7983ee14619bd71f2ea4001c2c6dba9ab79da4920d5c",
    "role": "admin"
  }
}
```

## 2. créer un compte utilisateur existe

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/signup`. The request body is a JSON object: `{ "email": "x", "password": "qsd" }`. The response status is `401 Unauthorized` with a message: `"message": "Utilisateur déjà enregistré", "user": { "email": "x", "password": "0b1b8e9fc13fa4c07a6d7983ee14619bd71f2ea4001c2c6dba9ab79da4920d5c", "role": "admin" }`.

POST `http://localhost:3000/signup`

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

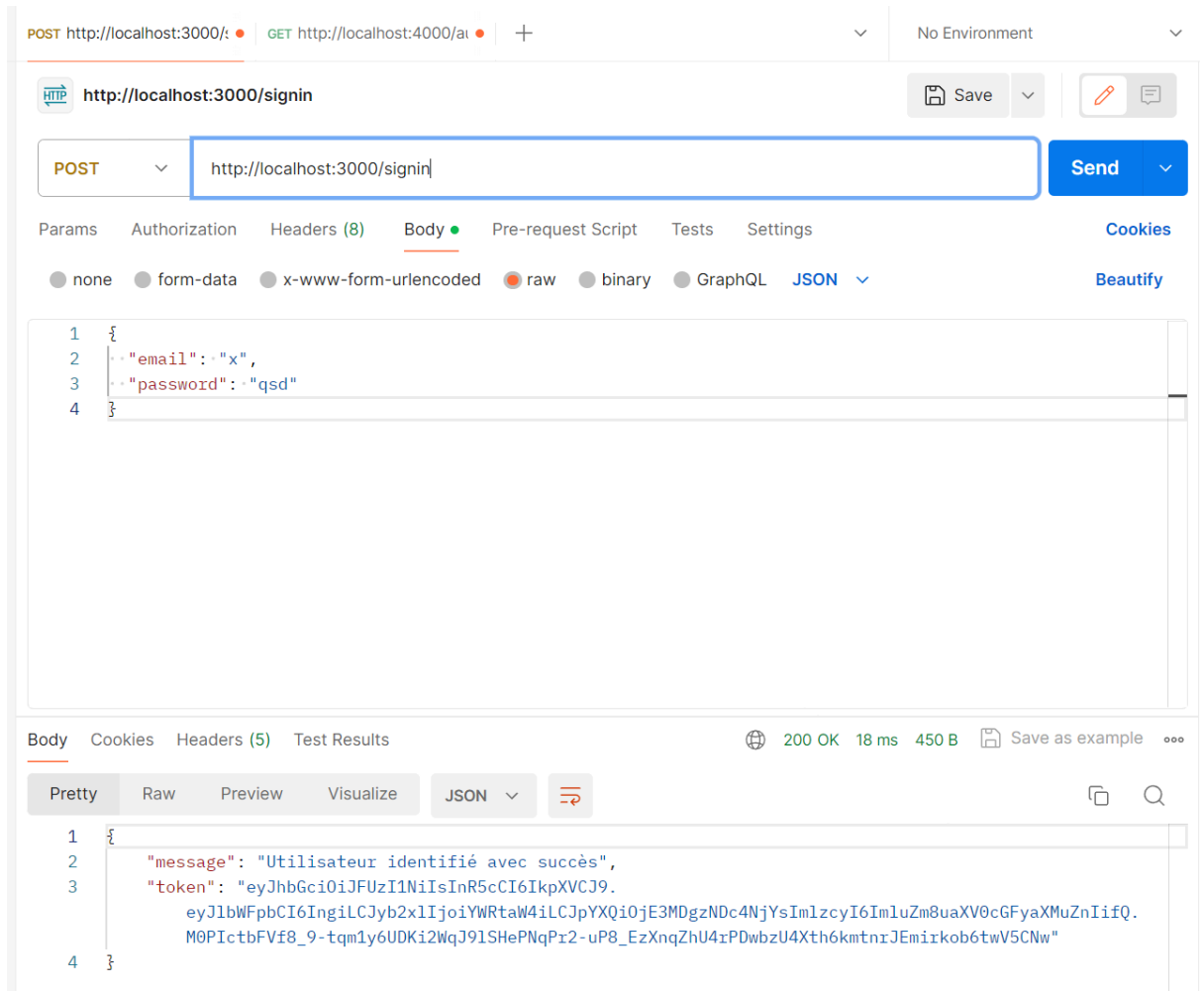
```
1 {
2   "email": "x",
3   "password": "qsd"
4 }
```

Body Cookies Headers (5) Test Results 401 Unauthorized 5 ms 340 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Utilisateur déjà enregistré",
3   "user": {
4     "email": "x",
5     "password": "0b1b8e9fc13fa4c07a6d7983ee14619bd71f2ea4001c2c6dba9ab79da4920d5c",
6     "role": "admin"
7   }
8 }
```

3. se connecter avec un compte utilisateur créé avec des informations correspondent



#### 4. se connecter avec mauvaise information

The screenshot shows the Postman interface for a POST request to `http://localhost:3000/signin`. The request body is a JSON object with `email: "xt"` and `password: "qsd"`. The response is a 401 Unauthorized status with a message: `"message": "Utilisateur non-identifié"`.

**Request Details:**

- Method: POST
- URL: `http://localhost:3000/signin`
- Body (JSON):

```
1 {
2   "email": "xt",
3   "password": "qsd"
4 }
```

**Response Details:**

- Status: 401 Unauthorized
- Time: 10 ms
- Size: 221 B
- Body (JSON):

```
1 {
2   "message": "Utilisateur non-identifié"
3 }
```

5. /auth avec bon token : le rôle contenu dans le jeton est admin, alors une clé message de valeur « Full access » sera ajoutée au retour.

The screenshot shows a REST client interface with the following details:

- Request:** GET `http://localhost:4000/auth`
- Authorization:** Type: Bearer, Token: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1`
- Response:** 200 OK, 34 ms, 238 B. The response body is a JSON object:

```
1 {
2   "user": {
3     "email": "x",
4     "role": "admin"
5   },
6   "accessMessage": "Full access"
7 }
```



## 6. /auth avec mauvais token

The screenshot shows a REST client interface with the following details:

- Request:** GET `http://localhost:4000/auth`
- Authorization:** Bearer `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...`
- Response:** 401 Unauthorized, 12 ms, 289 B
- Response Body (JSON):**

```
{  "statusCode": 401,  "error": "Unauthorized",  "message": "\"ES256\" signatures must be \"64\" bytes, saw \"63\""}}
```

## 7. créer un nouveau utilisateur rôle utilisateur

POST http://localhost:3000/ GET http://localhost:4000/api +

No Environment

HTTP http://localhost:3000/signup Save

POST http://localhost:3000/signup Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "email": "xt",
3   "password": "qsd"
4 }
```

Body Cookies Headers (5) Test Results 200 OK 5 ms 346 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Utilisateur enregistré avec succès",
3   "newUser": {
4     "email": "xt",
5     "password": "0b1b8e9fc13fa4c07a6d7983ee14619bd71f2ea4001c2c6dba9ab79da4920d5c",
6     "role": "utilisateur"
7   }
8 }
```

## 8. se connecter avec les informations rôle utilisateur

The screenshot displays a REST client interface with a POST request to `http://localhost:3000/signin`. The request body is a JSON object containing `email` and `password` fields. The response is a 200 OK status with a JSON body indicating a successful login and providing a long JWT token.

**Request:**

```
POST http://localhost:3000/signin
```

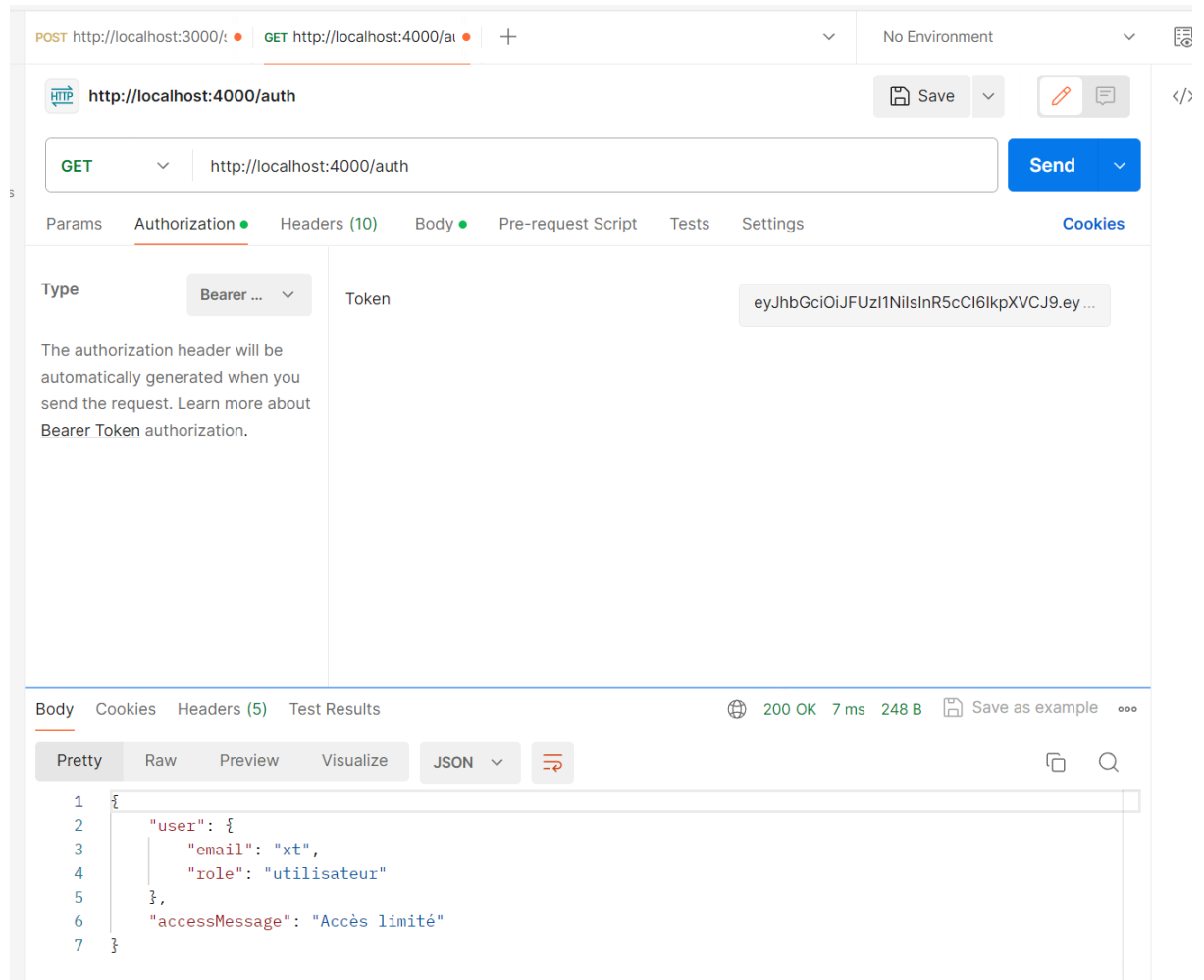
**Body:**

```
{
  "email": "xt",
  "password": "qsd"
}
```

**Response:**

```
200 OK 8 ms 459 B
{
  "message": "Utilisateur identifi  avec succ s",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFnZW50Iiwicm9sZSI6InV0aWxpc2F0ZXV5IiwiaWF0IjoxNzA4MzQ4MDU2LCJpc3MiOiJpbmZvLm11dHBhcm1zLmZyIn0.Upxe-GeG7_oL24WZkmvRPW5iQJwM08S4P_83FYsKEf0oL1sEdeq241f2T1LqUMA1V6bfdaQ_ksutaL00uXPcA"
}
```

9. /auth avec bon token : si le rôle est utilisateur, alors la valeur de message sera « Accès limité ».



### **vos difficultés, comment vous les avez surmontées**

J'ai essayé plusieurs fois le `req.jwtSign` et `req.jwtVerify` au début, avec des mauvais formats et mauvaise compréhension. À la fin j'ai enfin trouvé la documentation de `@fastify/jwt`

### **qu'est qu'on pourrait améliorer**

Ma compréhension de l'affichage retourné lors du test du certificat généré reste encore un peu floue pour moi.