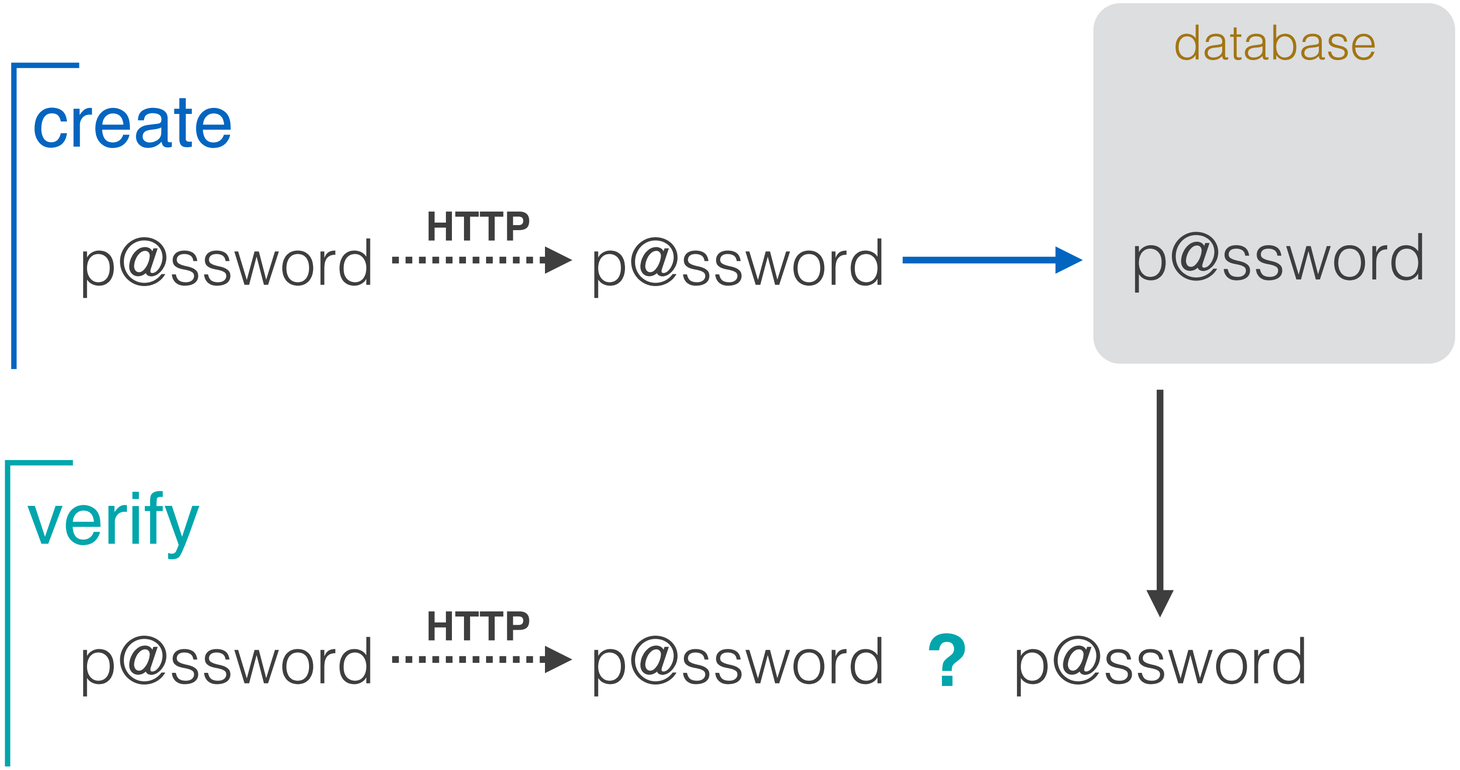
AUTHENTICATION DATA FLOW

I am who I say I am

AAA

- **Authentication**
  - “this person is who they say they are”
- **Authorization**
  - “this person is allowed to do X, Y, Z”
- **Accounting**
  - “who the heck is using all our bandwidth?”

SIGNUP/LOGIN



STAYING LOGGED IN

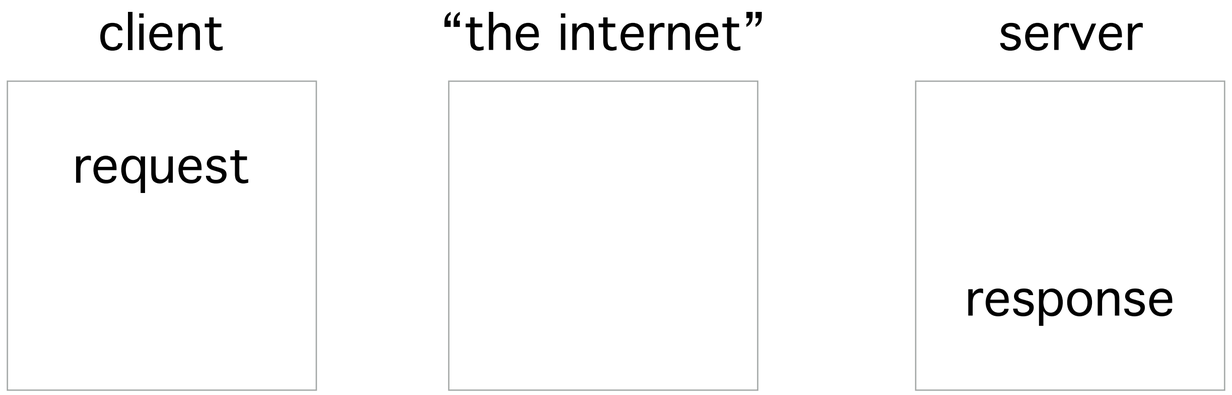
THE PROBLEM

What if we want to store some information about each client/server relationship?

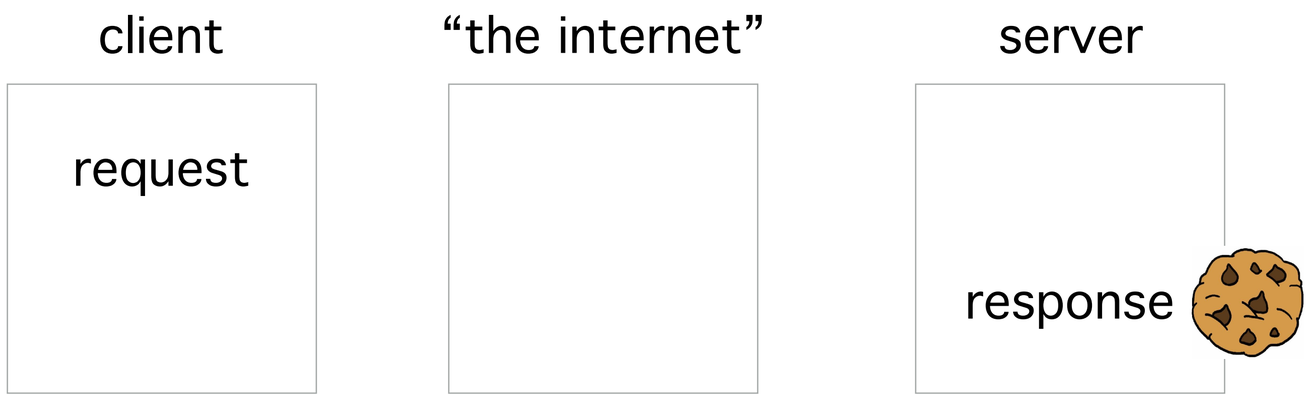

COOKIE!

= small text file

HTTP



WITH COOKIES



WITH COOKIES

client

“the internet”

server

WITH COOKIES

client

request

“the internet”

server

response

Do something with cookie

COOKIES & SESSIONS

No need to authenticate for every request

SESSIONS

client

“the internet”

server

request

response

Make server session

sessions[id] = {}

SESSIONS

client

“the internet”

server

SESSIONS

client

request

“the internet”

server

response

Look up session

req.session = sessions[id]

COOKIES & SESSIONS

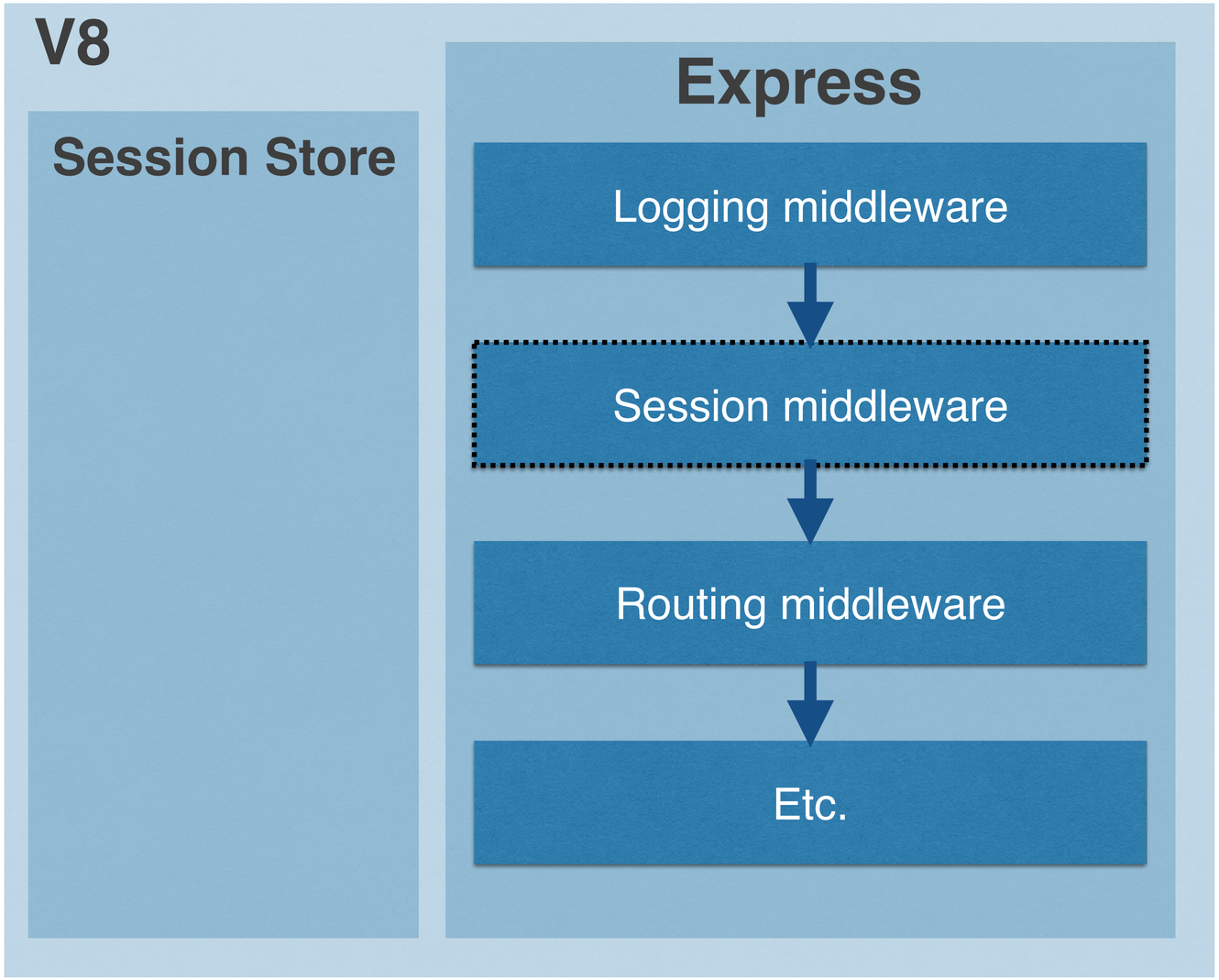
- Server gives client a cookie with ID only
- Client keeps cookie and sends with all requests
- Server loads client-specific session (stored in RAM)

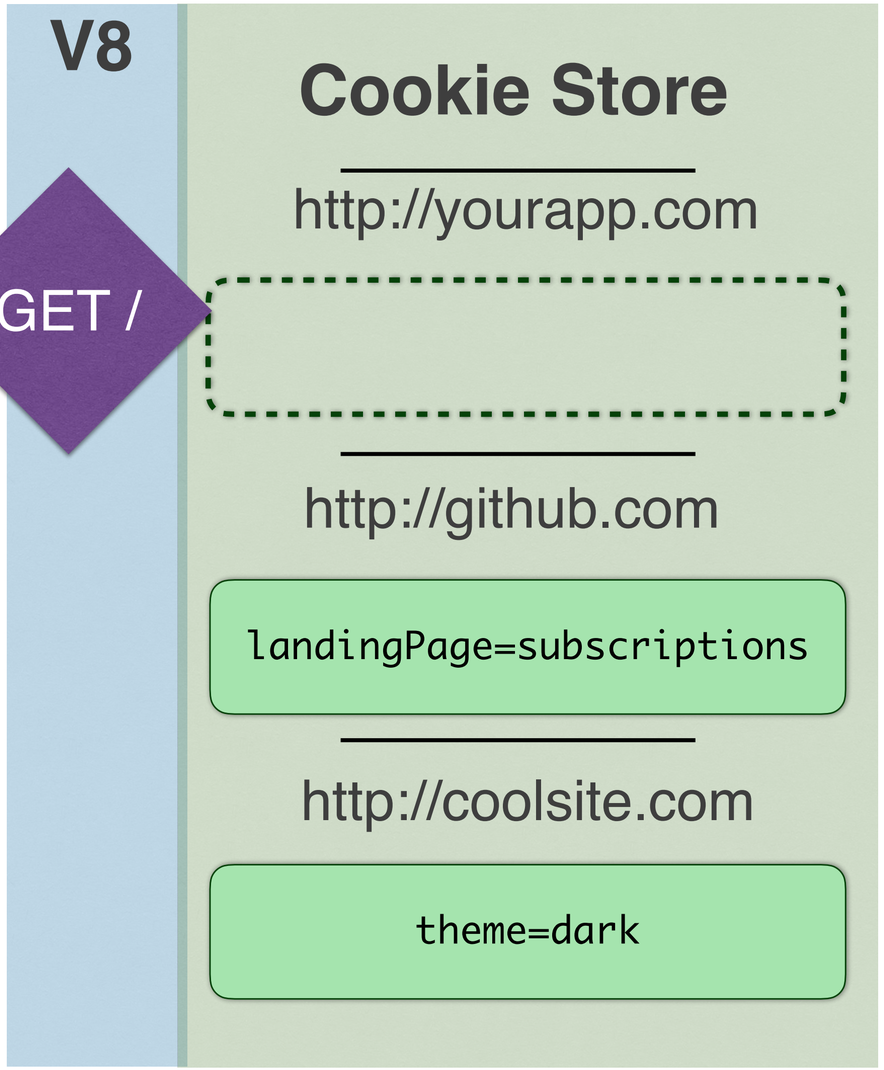

SESSIONS
IN
DETAIL

Server (Backend)

Internet (HTTP)

Client (Browser)





http://yourapp.com

HTTP REQUEST

```
GET / HTTP/1.1
Host: yourapp.com
Connection: keep-alive
Accept: text/html
User-Agent: Chrome/
48.0.2564.116
…etc…
```

…no cookie info (yet)!

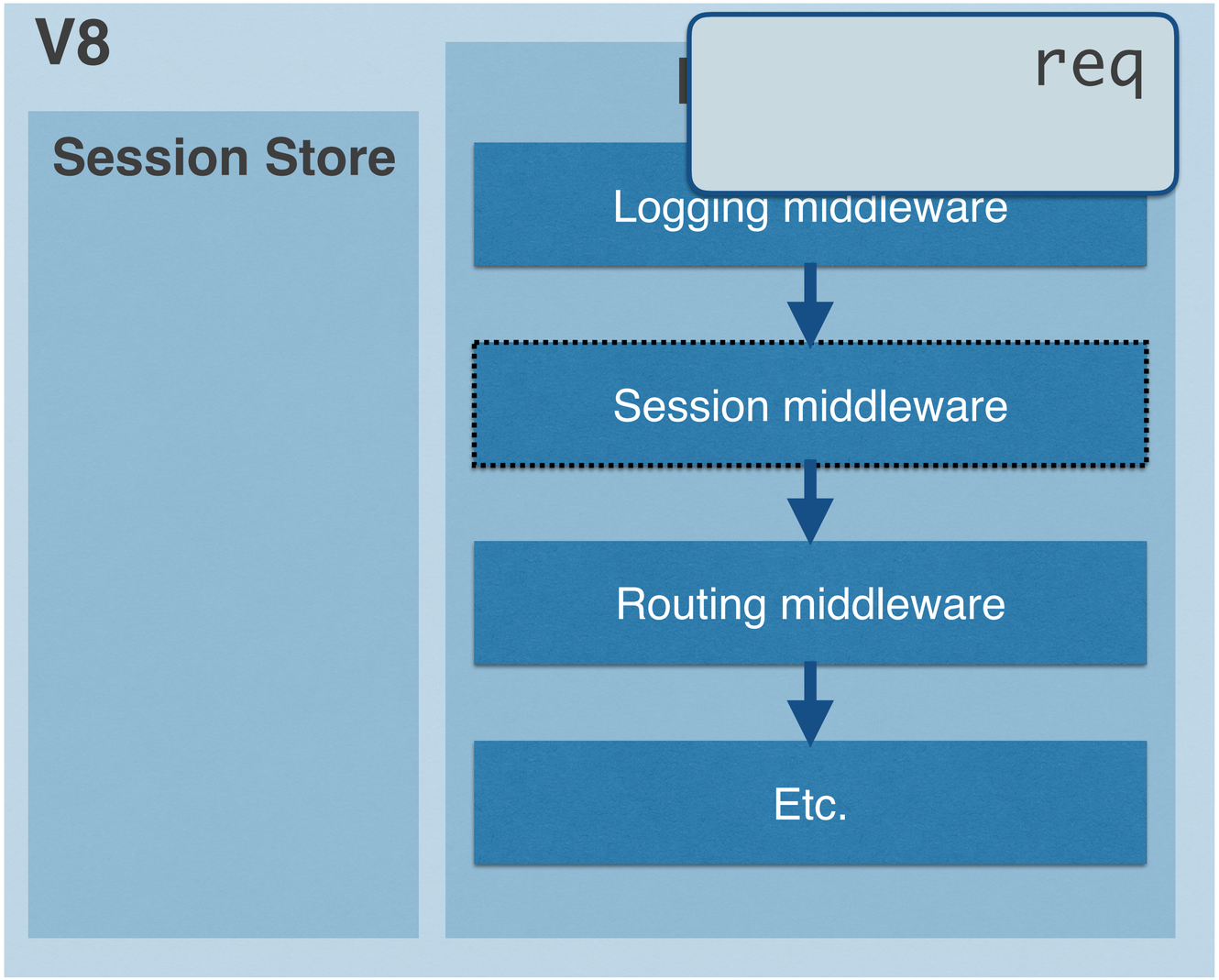
(headers)

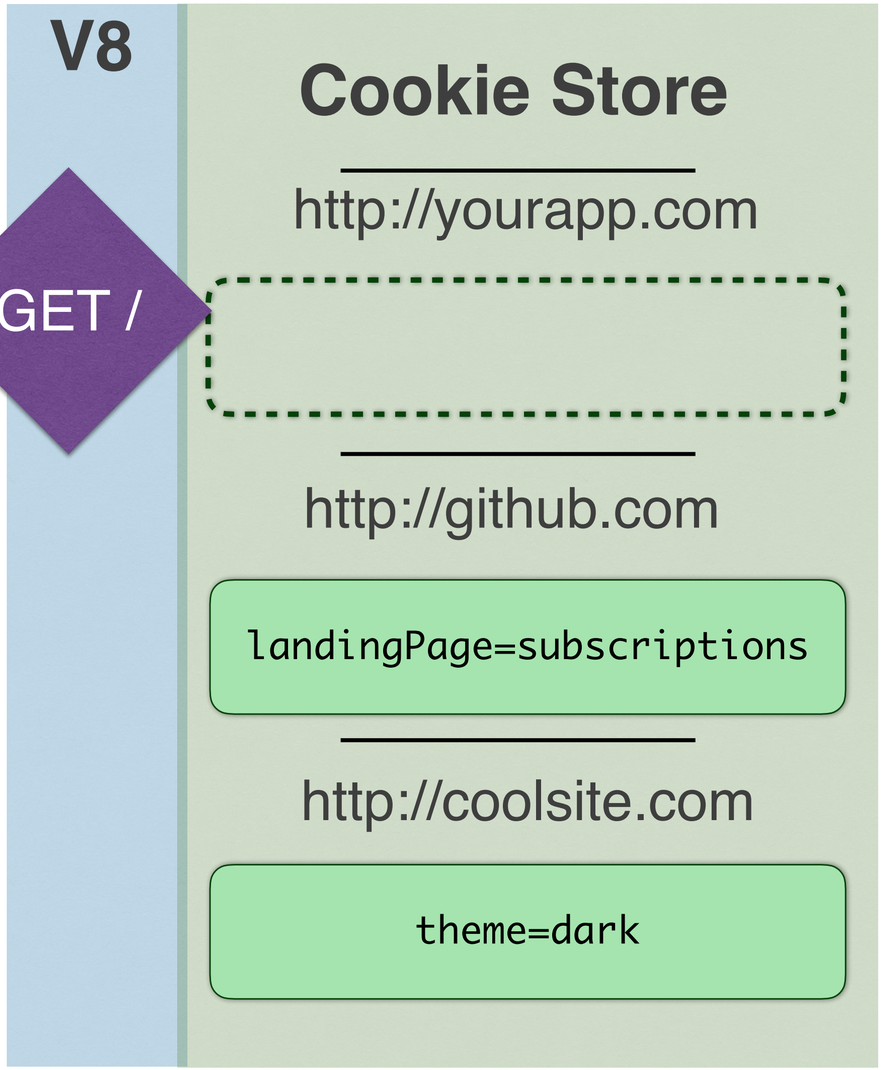
(body)

Server (Backend)

Internet (HTTP)

Client (Browser)





http://yourapp.com

HTTP REQUEST

```
GET / HTTP/1.1
Host: yourapp.com
Connection: keep-alive
Accept: text/html
User-Agent: Chrome/
48.0.2564.116
…etc…
```

(headers)

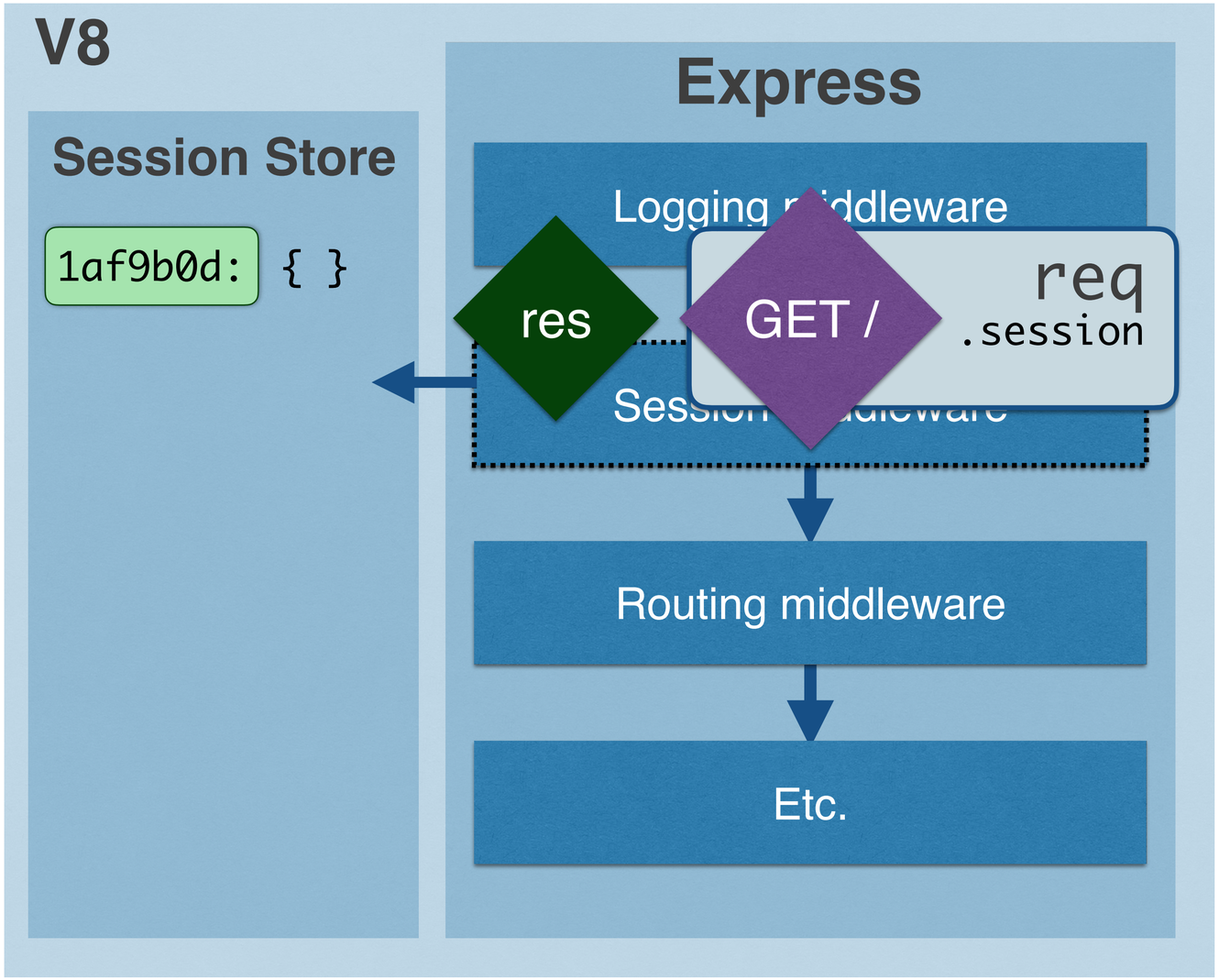
(body)

still no cookie info…
session middleware:
"let's make a session!"

Server (Backend)

Internet (HTTP)

Client (Browser)



http://yourapp.com

HTTP RESPONSE

```
HTTP/1.x 200 OK
Transfer-Encoding: chunked
Date: Mon, 22 Feb 2016 18:30:00 GMT
Content-Type: text/html
Content-Encoding: gzip
set-cookie: myUid=1af9b0d
```

(headers)

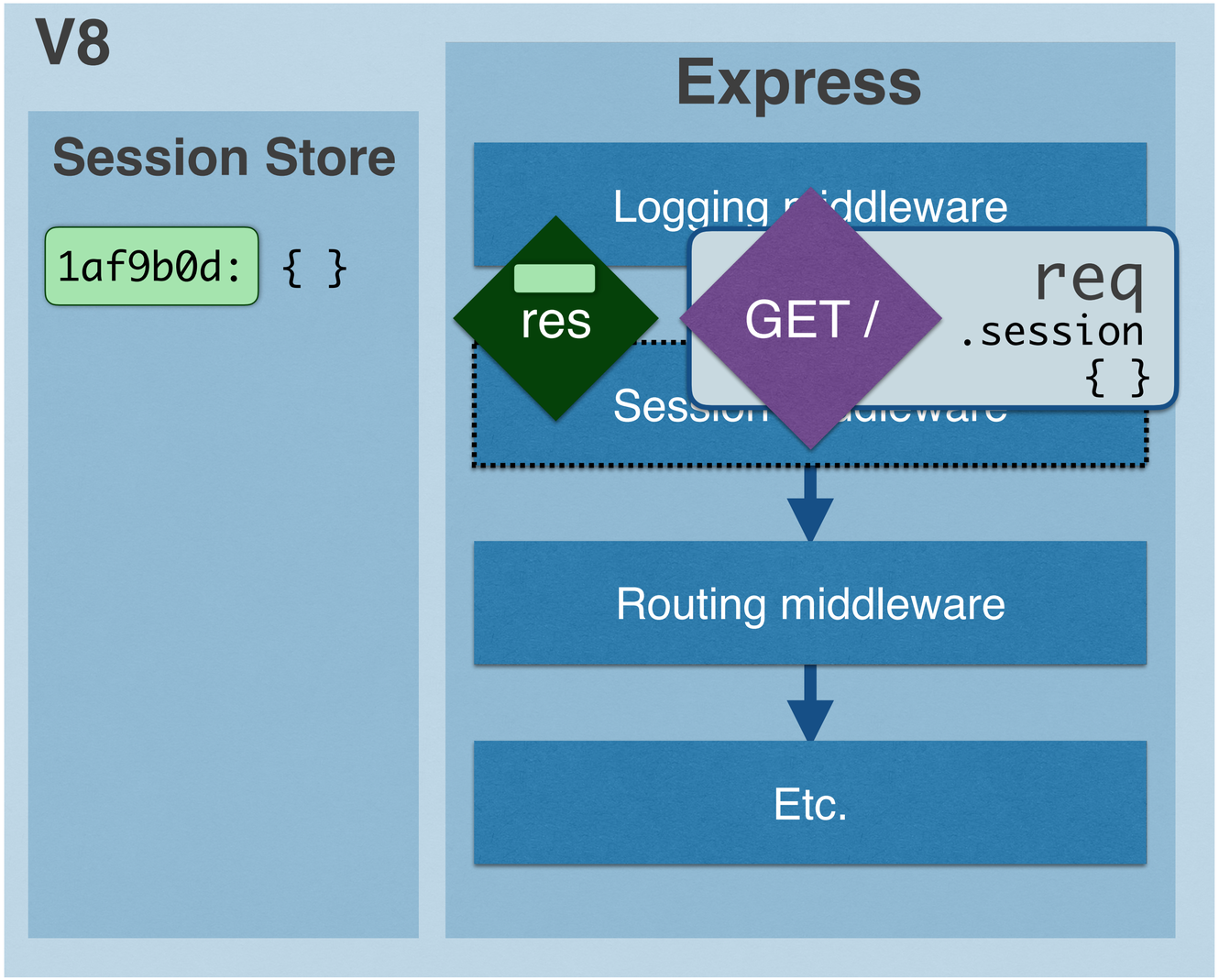
(body, after routing is done)

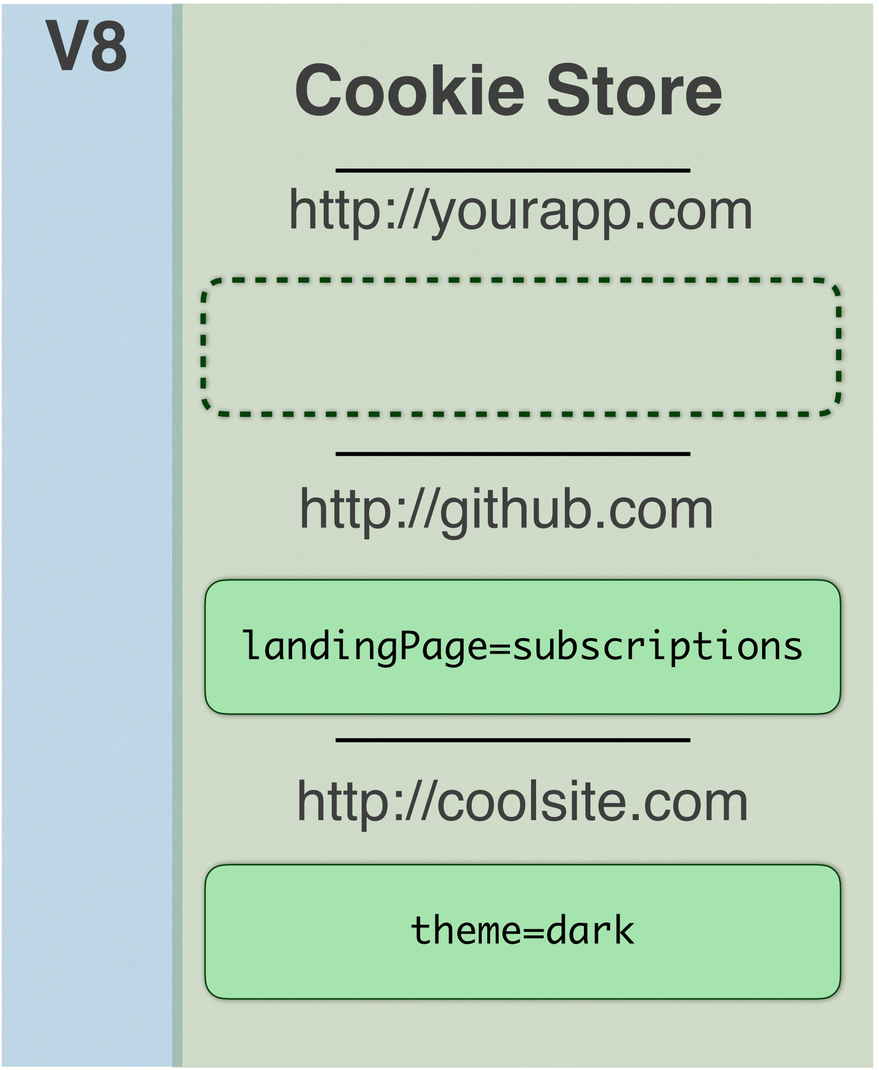
```
<!DOCTYPE html>
<html>
  <head>
    <title>Your Sweet App</title>
    <script src="/js/main.js"></script>
  </head> …etc.
```

Server (Backend)

Internet (HTTP)

Client (Browser)



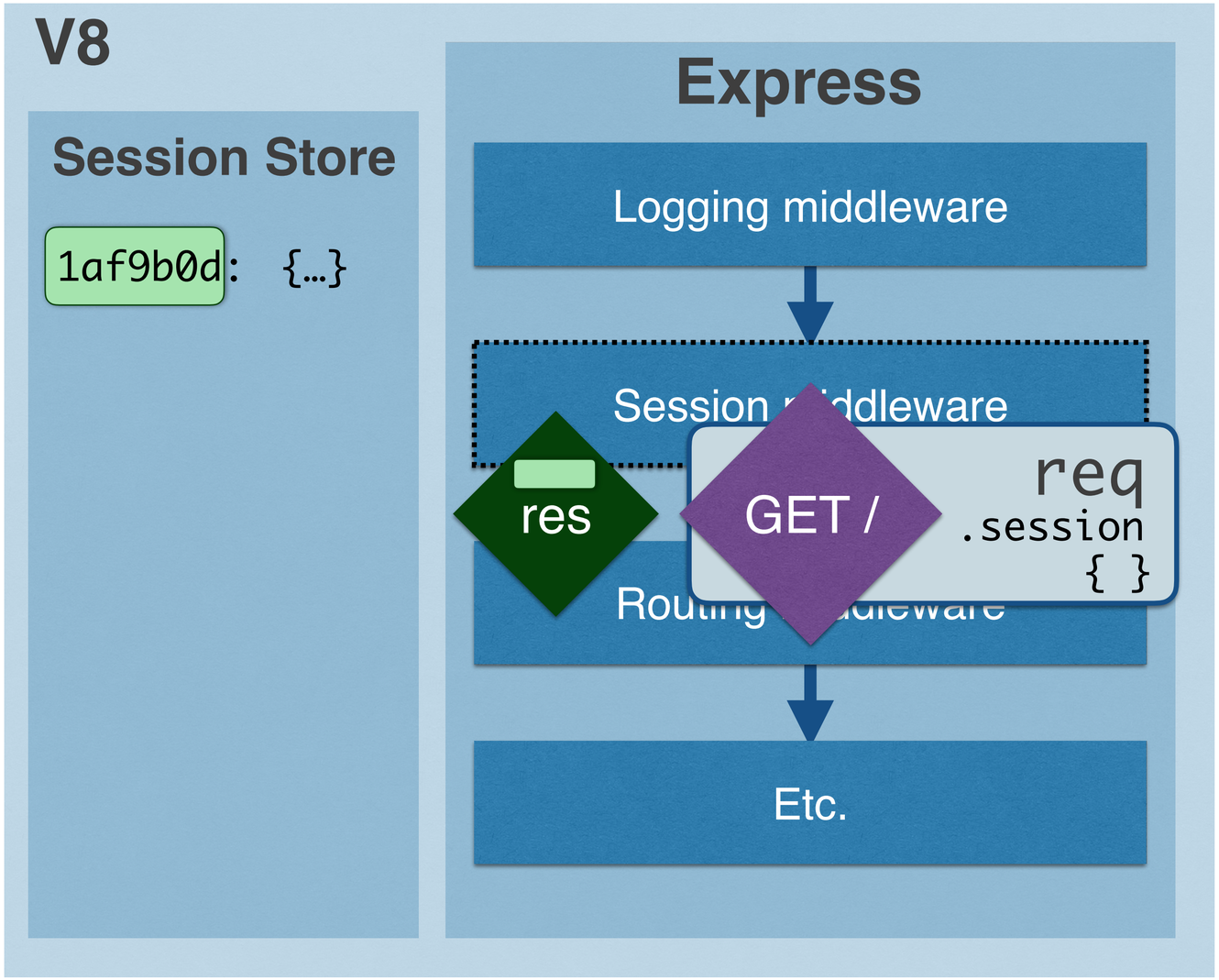


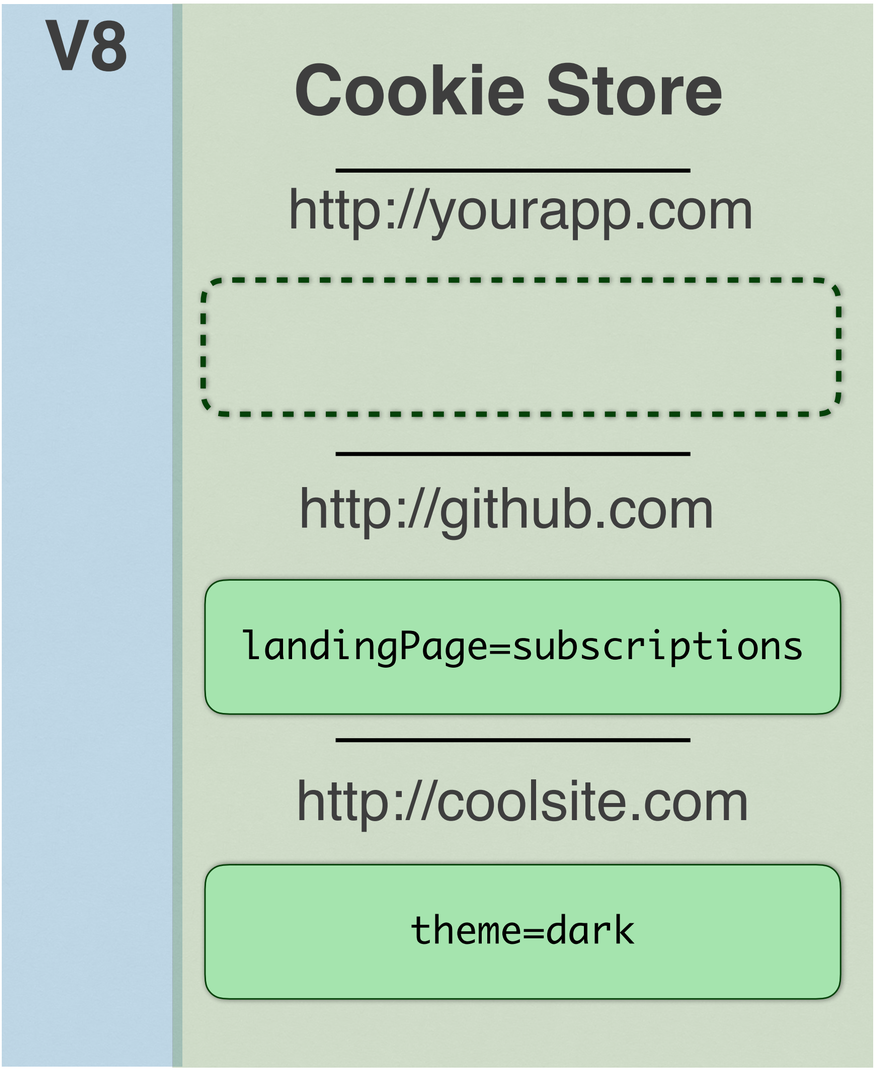
http://yourapp.com

Server (Backend)

Internet (HTTP)

Client (Browser)





http://yourapp.com

HTTP RESPONSE

(headers)

```
HTTP/1.x 200 OK
Transfer-Encoding: chunked
Date: Mon, 22 Feb 2016 18:30:00 GMT
Content-Type: text/html
Content-Encoding: gzip
set-cookie: myUid=1af9b0d
```

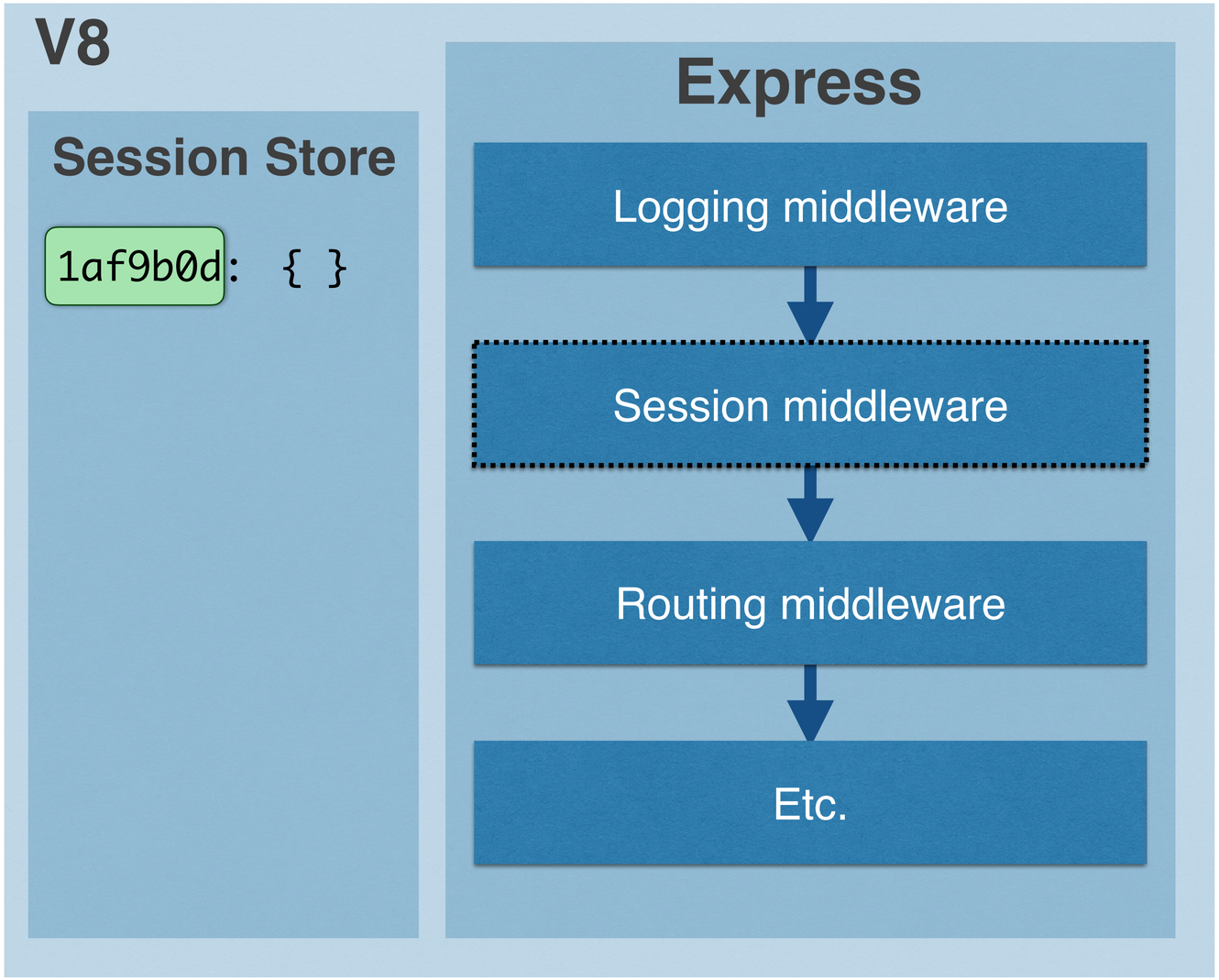
(body, after routing is done)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Your Sweet App</title>
    <script src="/js/main.js"></script>
  </head> …etc.
```

Server (Backend)

Internet (HTTP)

Client (Browser)

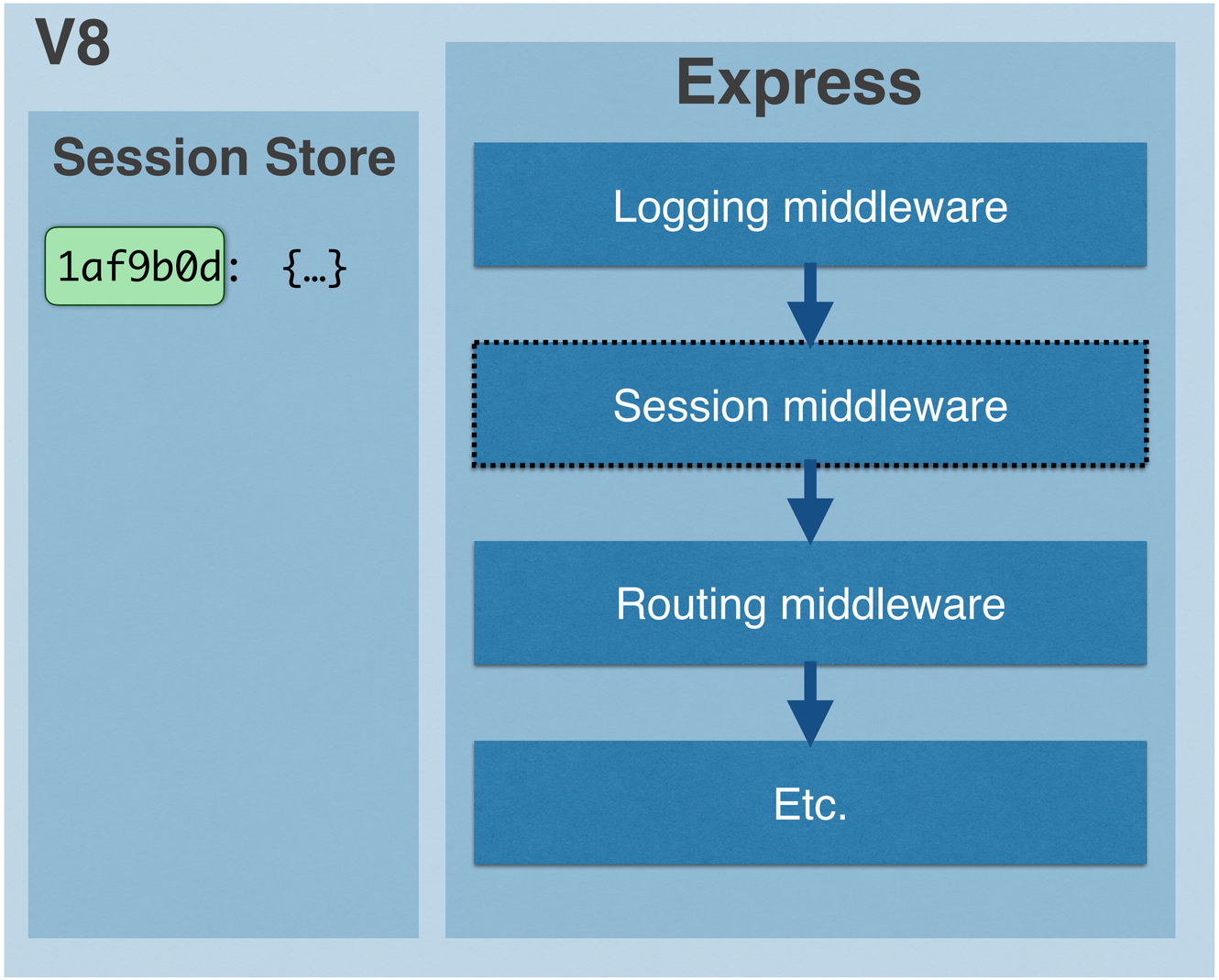


http://yourapp.com

Server (Backend)

Internet (HTTP)

Client (Browser)



http://yourapp.com

HTTP REQUEST

```
GET / HTTP/1.1
Host: yourapp.com
Connection: keep-alive
Accept: text/html
User-Agent: Chrome/
48.0.2564.116
Cookie: myUid=1af9b0d
```

…attached cookie!

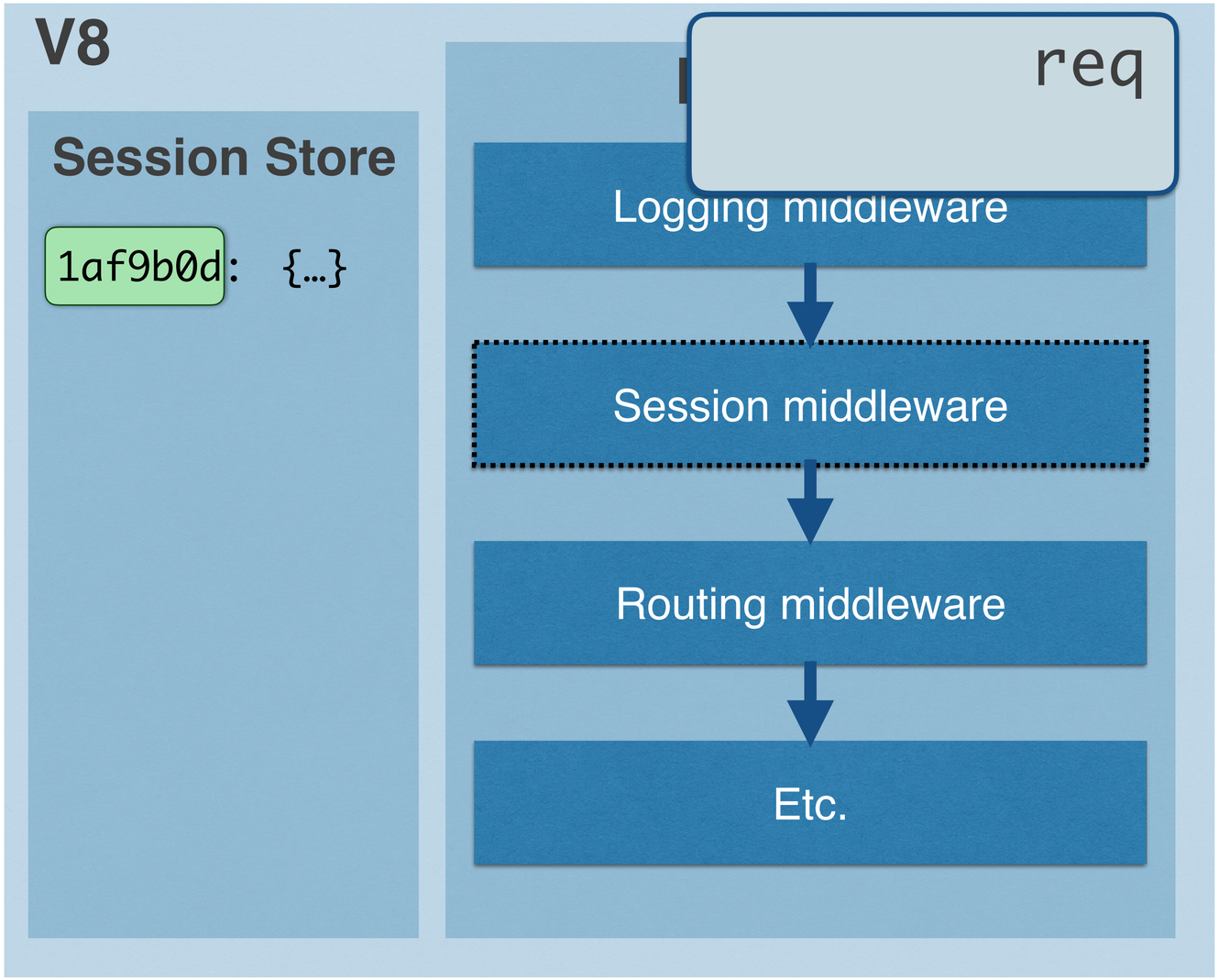
(headers)

(body)

Server (Backend)

Internet (HTTP)

Client (Browser)



http://yourapp.com

HTTP REQUEST

```
GET / HTTP/1.1
Host: yourapp.com
Connection: keep-alive
Accept: text/html
User-Agent: Chrome/
48.0.2564.116
Cookie: myUid=1af9b0d
```

attached cookie, so session middleware: let's look up the session!

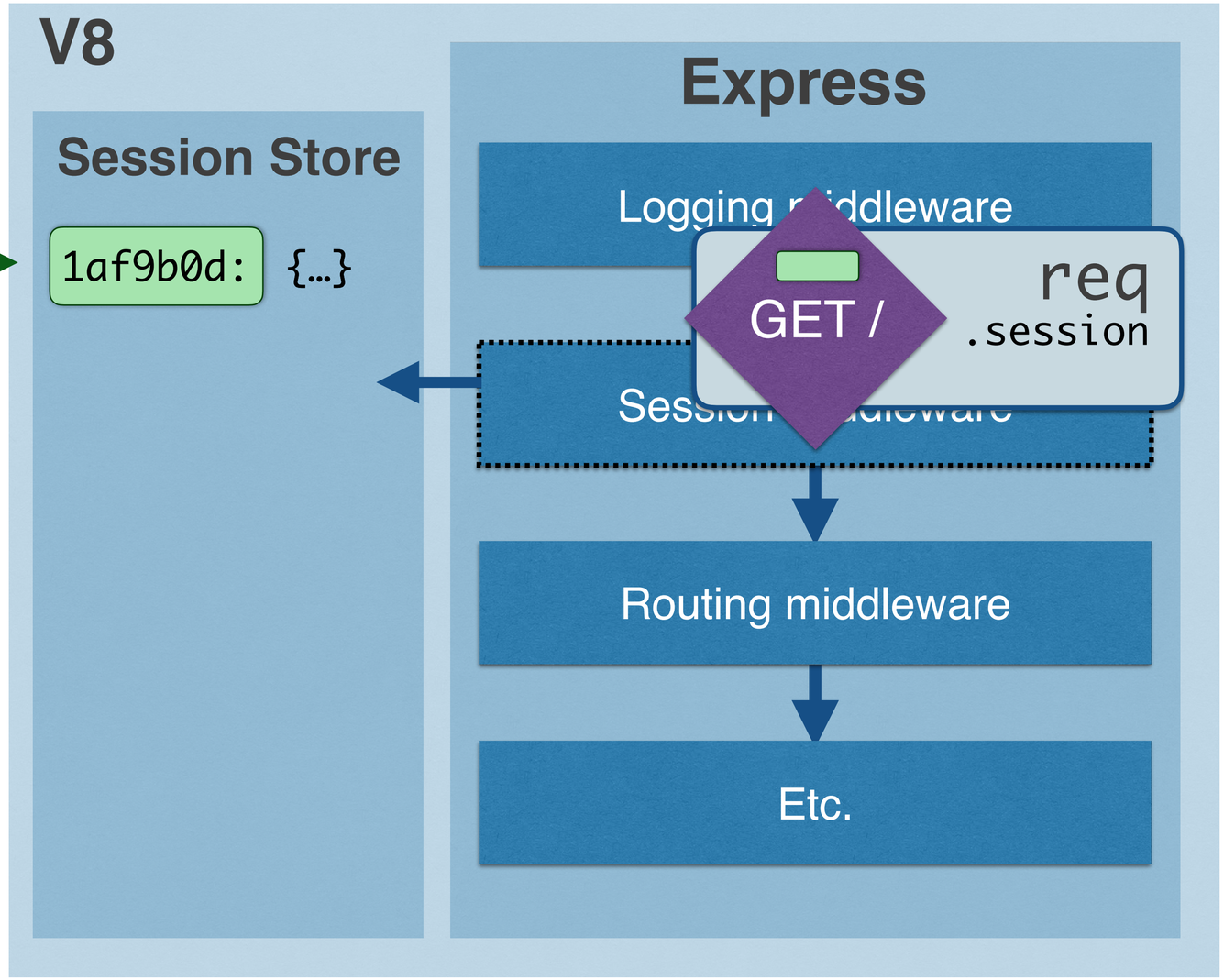
(headers)

(body)

Server (Backend)

Internet (HTTP)

Client (Browser)

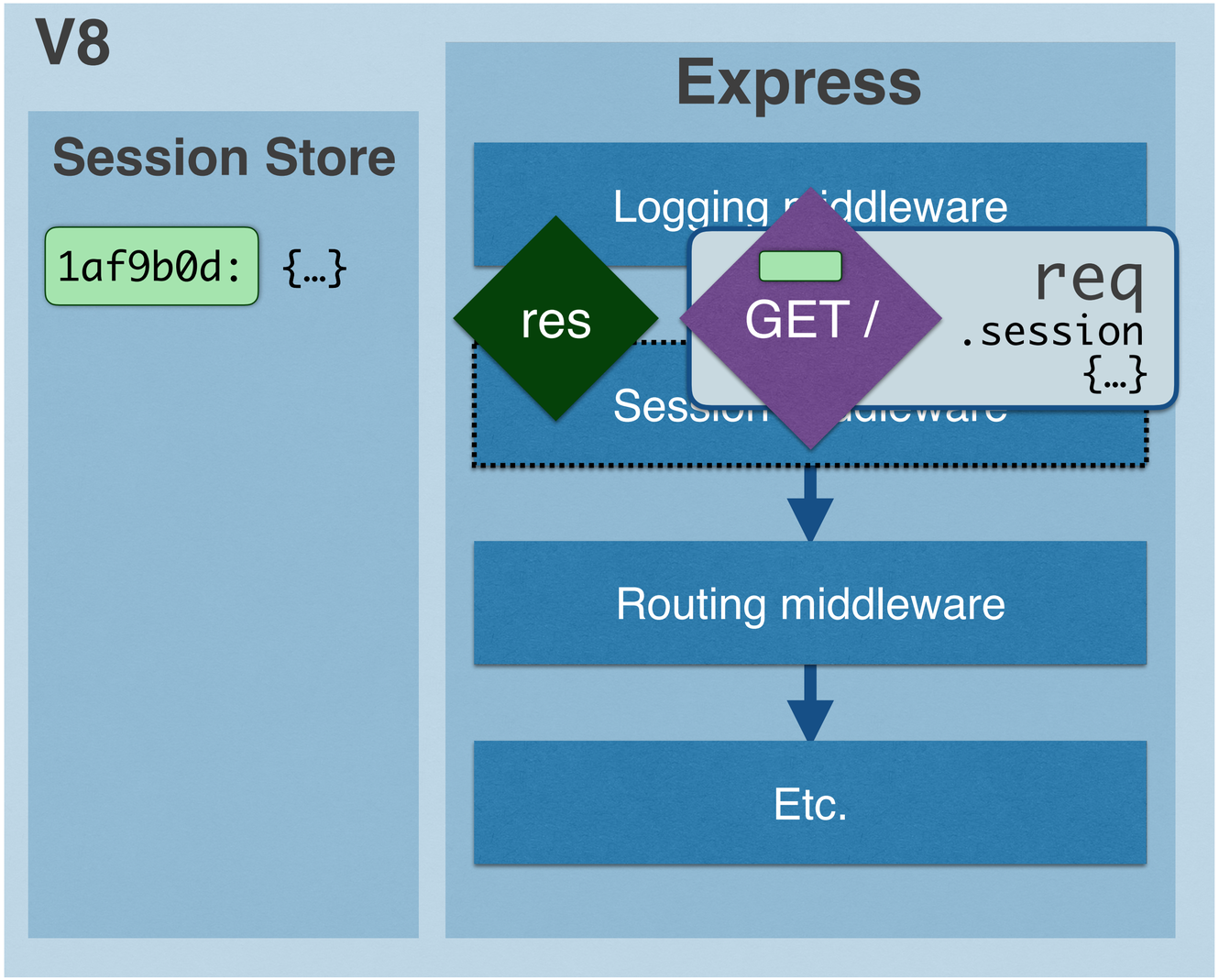


http://yourapp.com

Server (Backend)

Internet (HTTP)

Client (Browser)



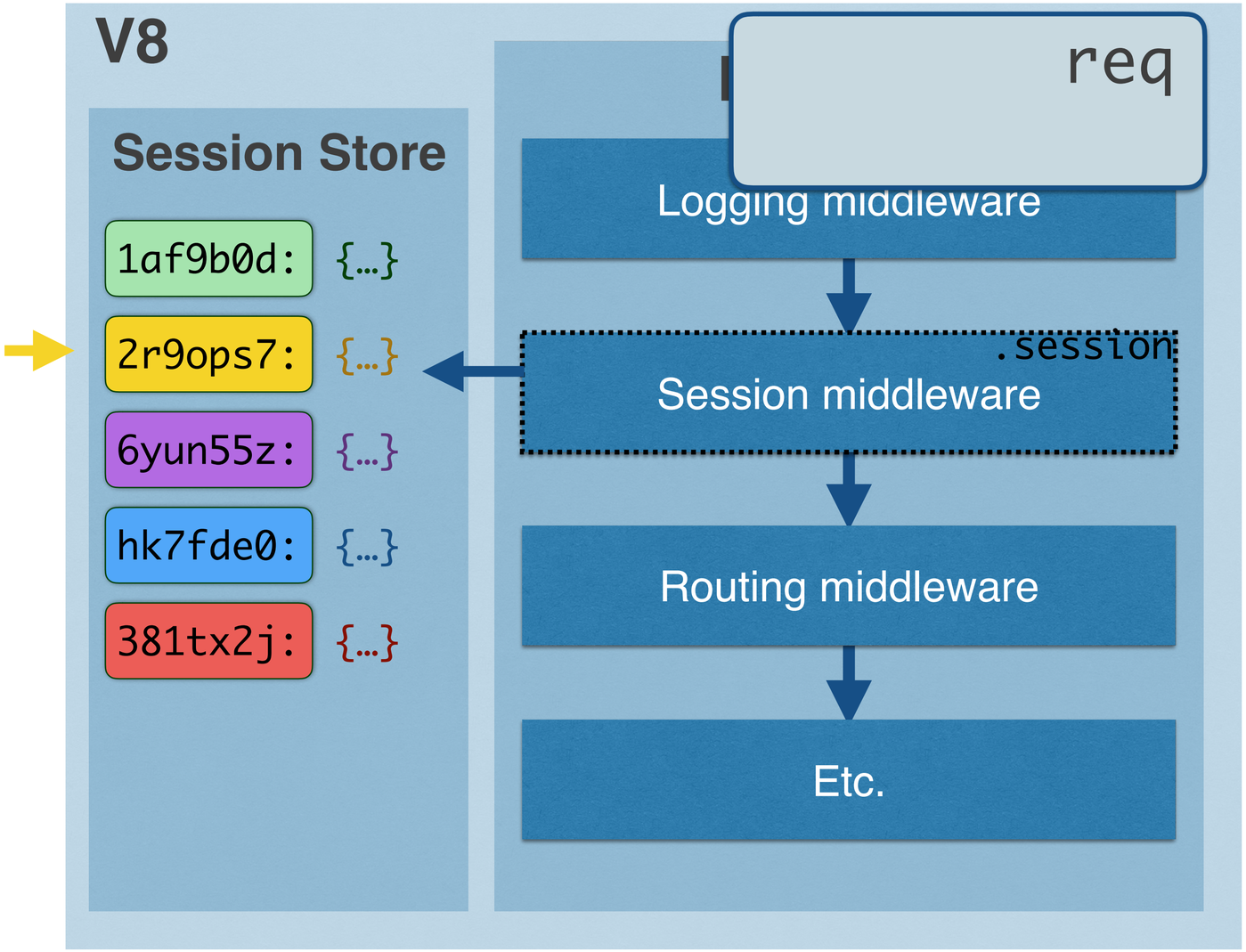
http://yourapp.com

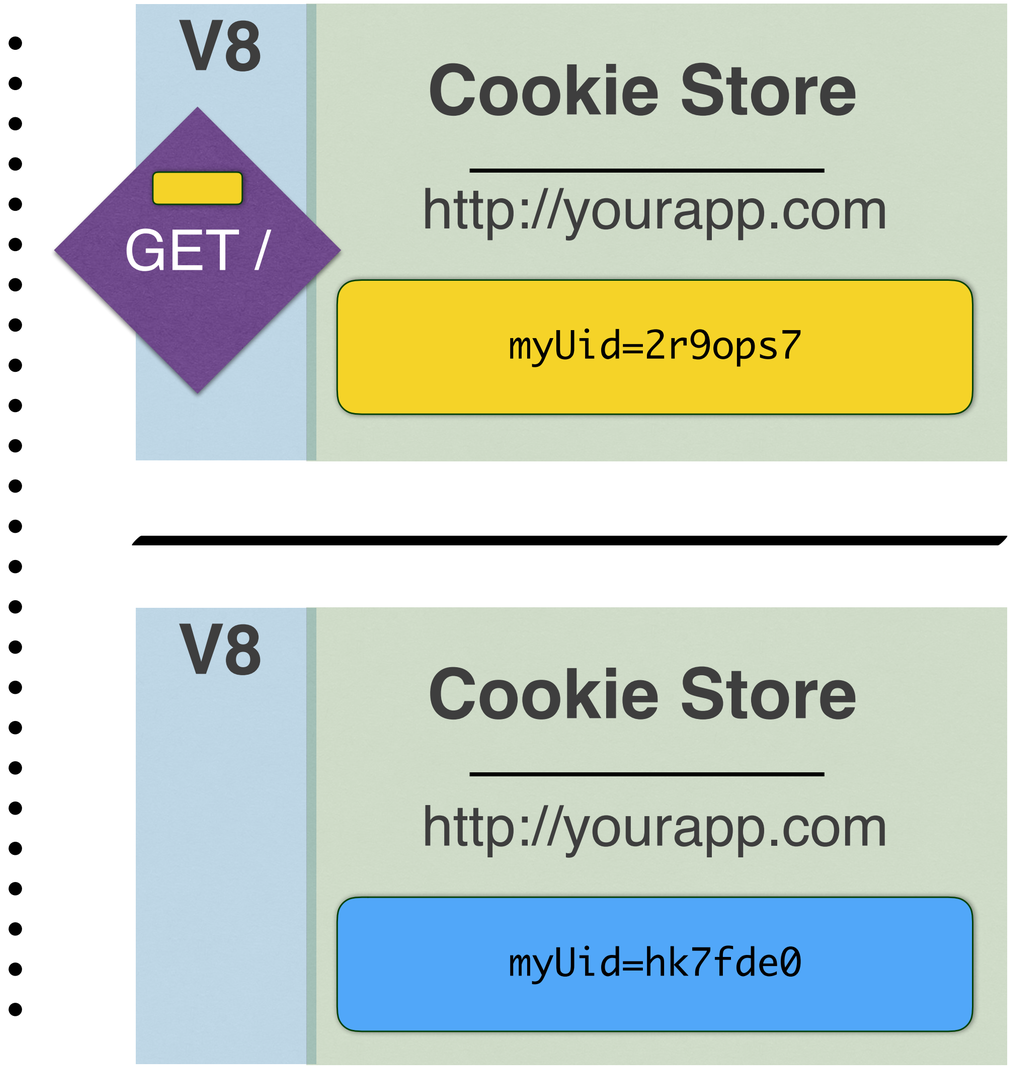
Multiple Sessions

Server (Backend)

Internet (HTTP)

Client 1 (Browser)





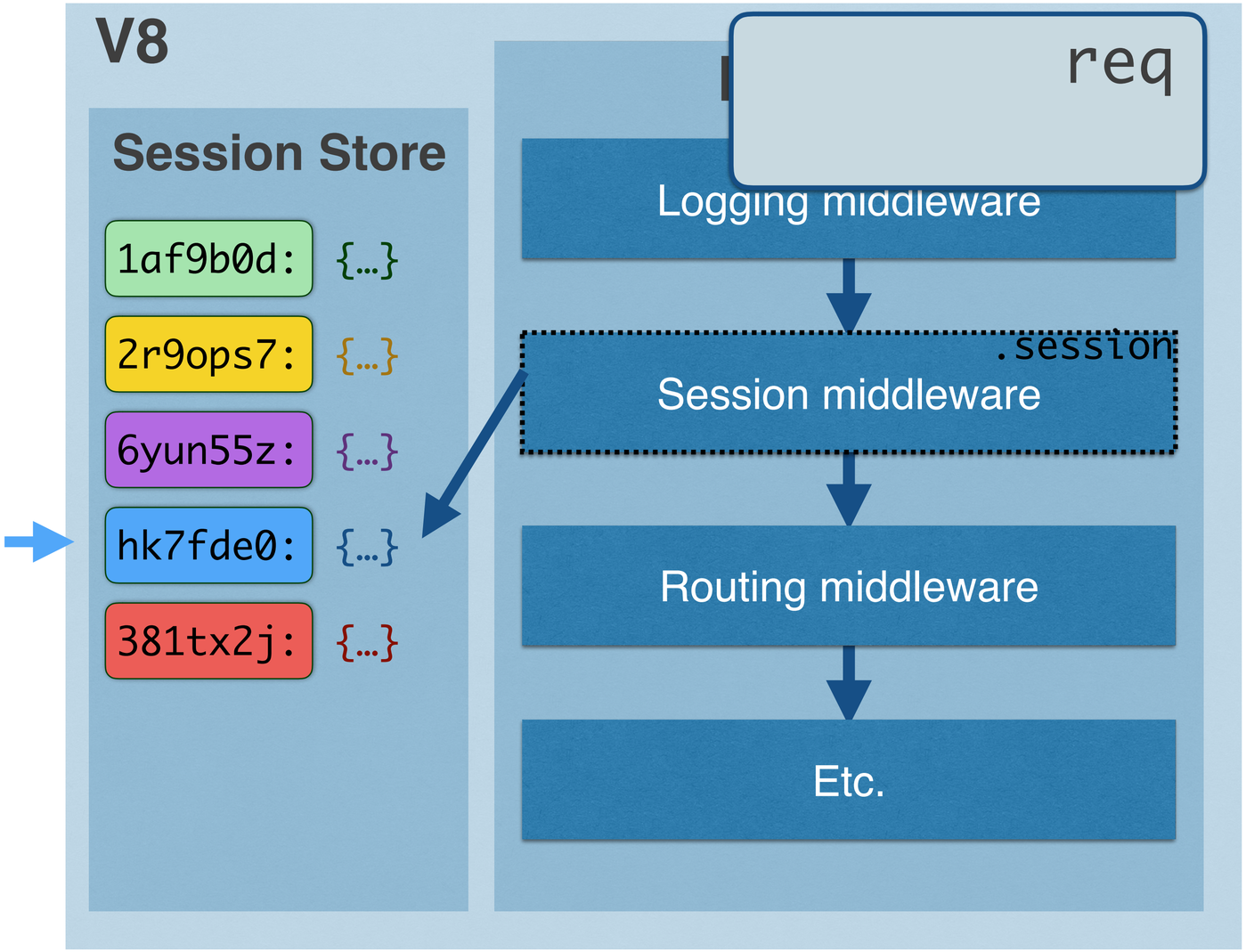
http://yourapp.com

Client 2 (Browser)

Server (Backend)

Internet (HTTP)

Client 1 (Browser)



http://yourapp.com

Client 2 (Browser)


OAUTH
OAUTH

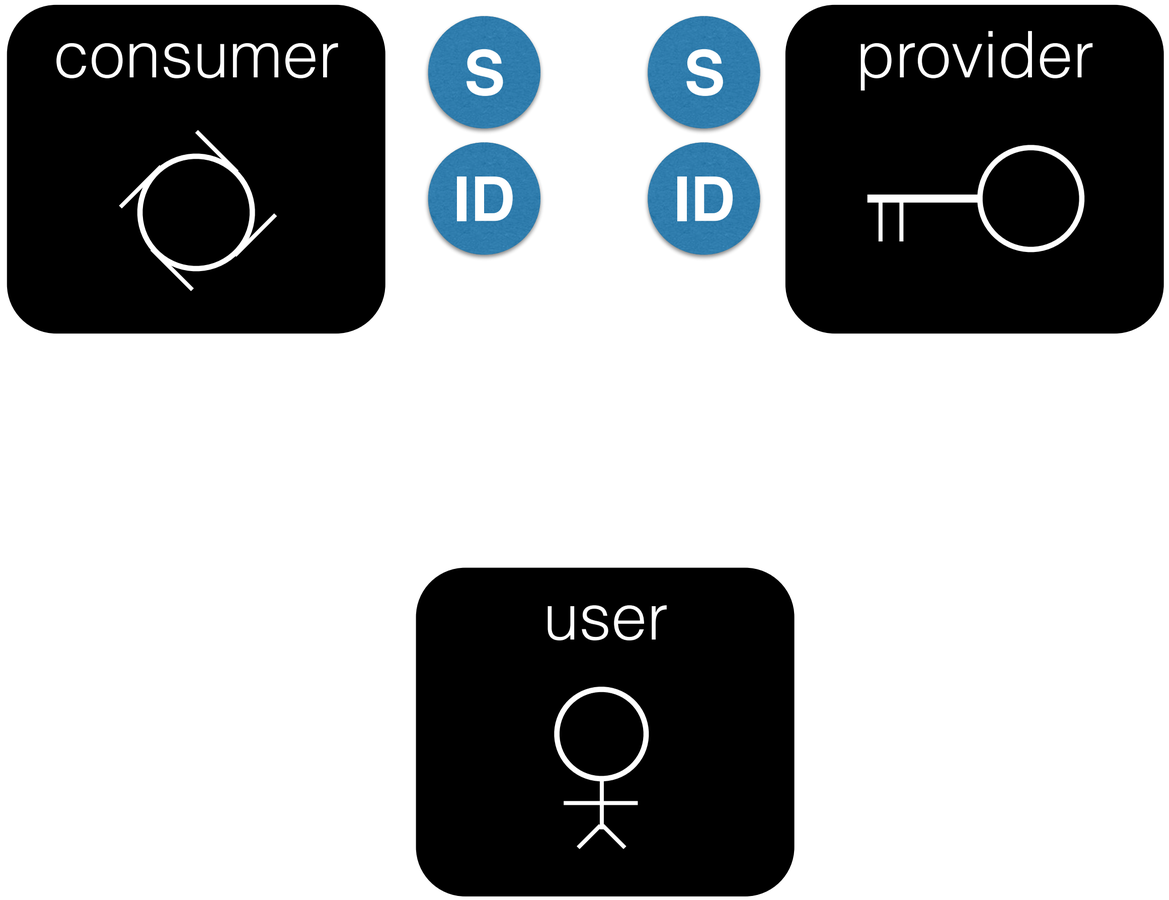
OAUTH

Standard protocol for auth piggybacking

Your Application	User	3rd Party Authority
“consumer”	“user”	“provider”

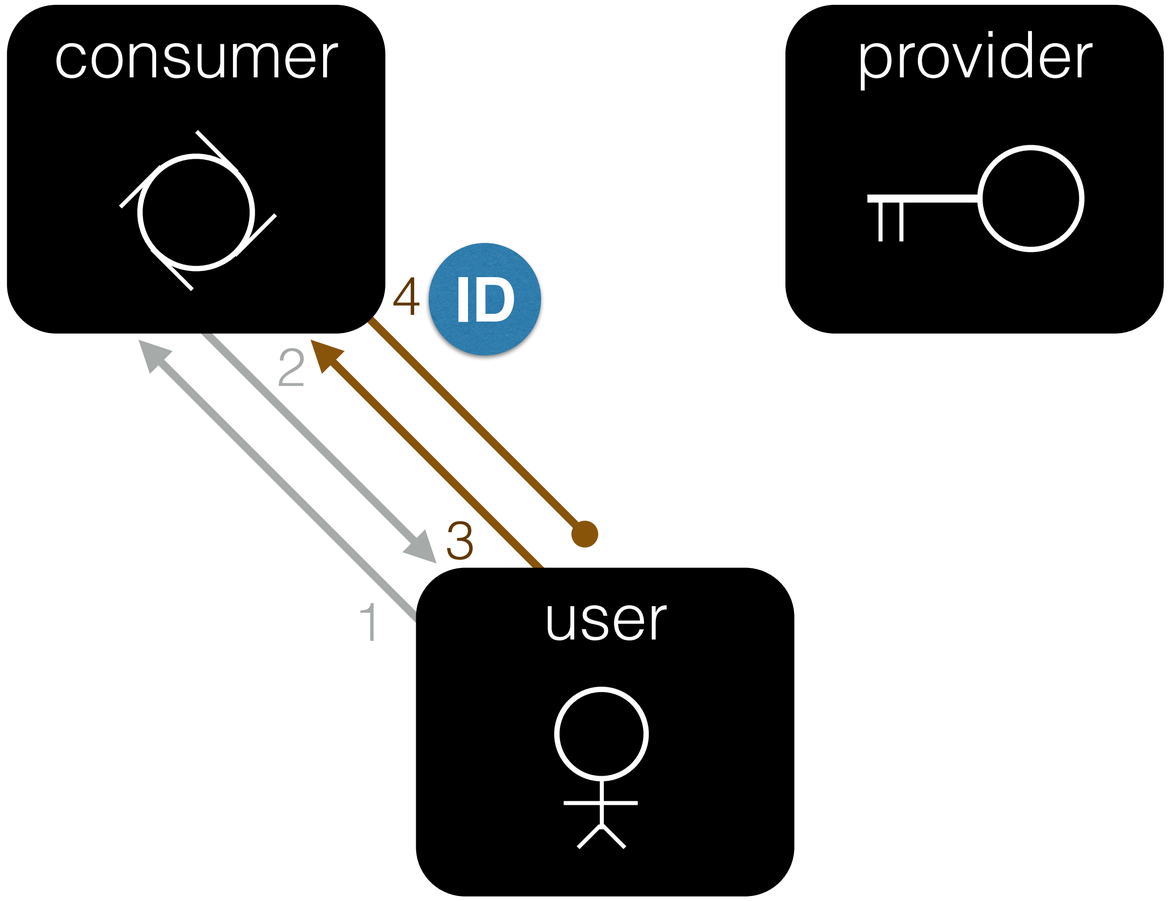
OAUTH - PREP

- Consumer app has a registered dev account with the provider.
- Provider gives consumer a public client ID and private client Secret.
- Both services hold on to these credentials so the consumer can prove who it is to the provider.



OAUTH - PETITION

1. Request: load login
2. Response: rendered login
3. Request: login through provider
4. Response: redirect to provider



http://provider.com/oauth/authorize?client_id=123&callback_url=consumer.com/confirm&scope=read

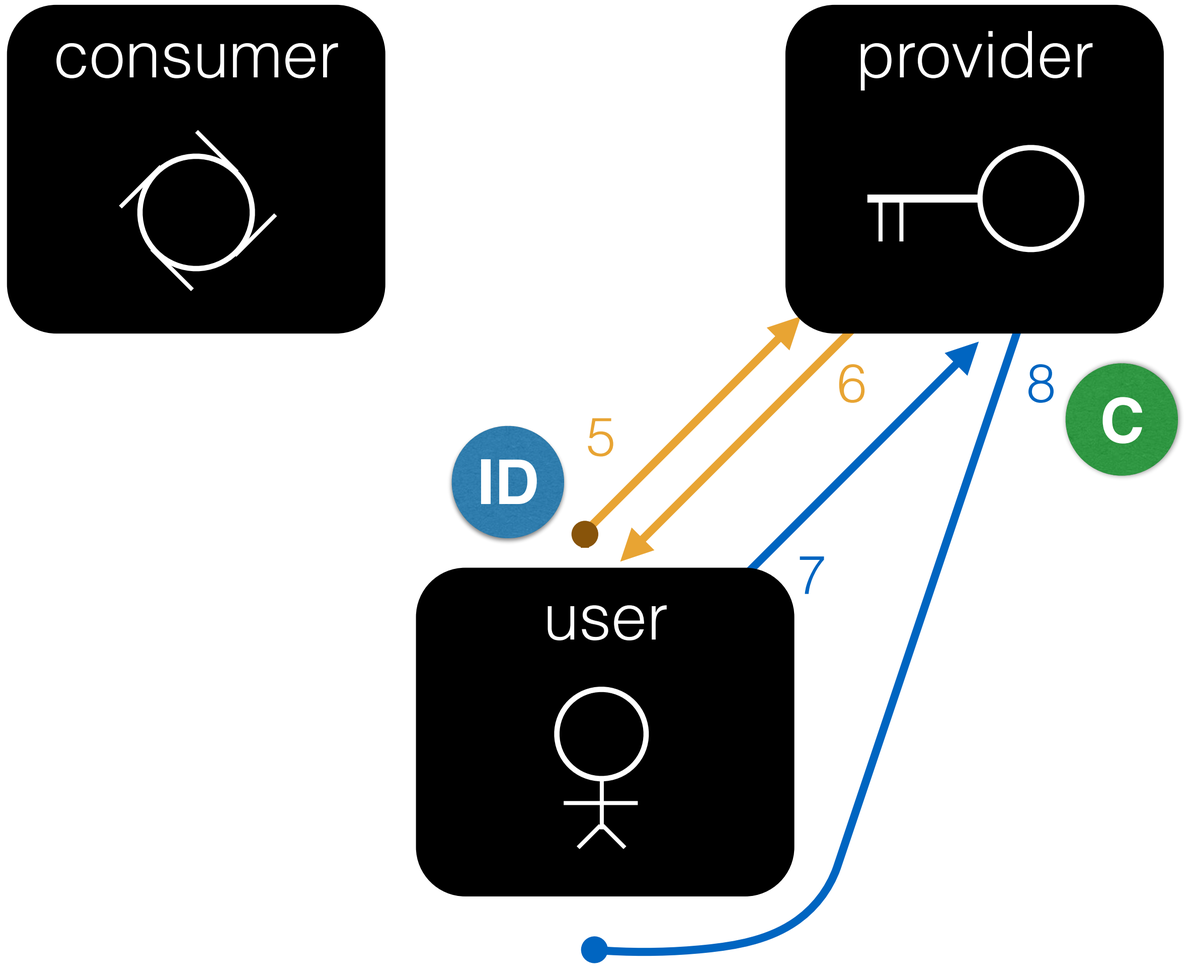
OAUTH - USER AUTHENTICATION

5.Request: load login

6.Response: rendered login

7.Request: login to provider

8.Response: redirect callback URL

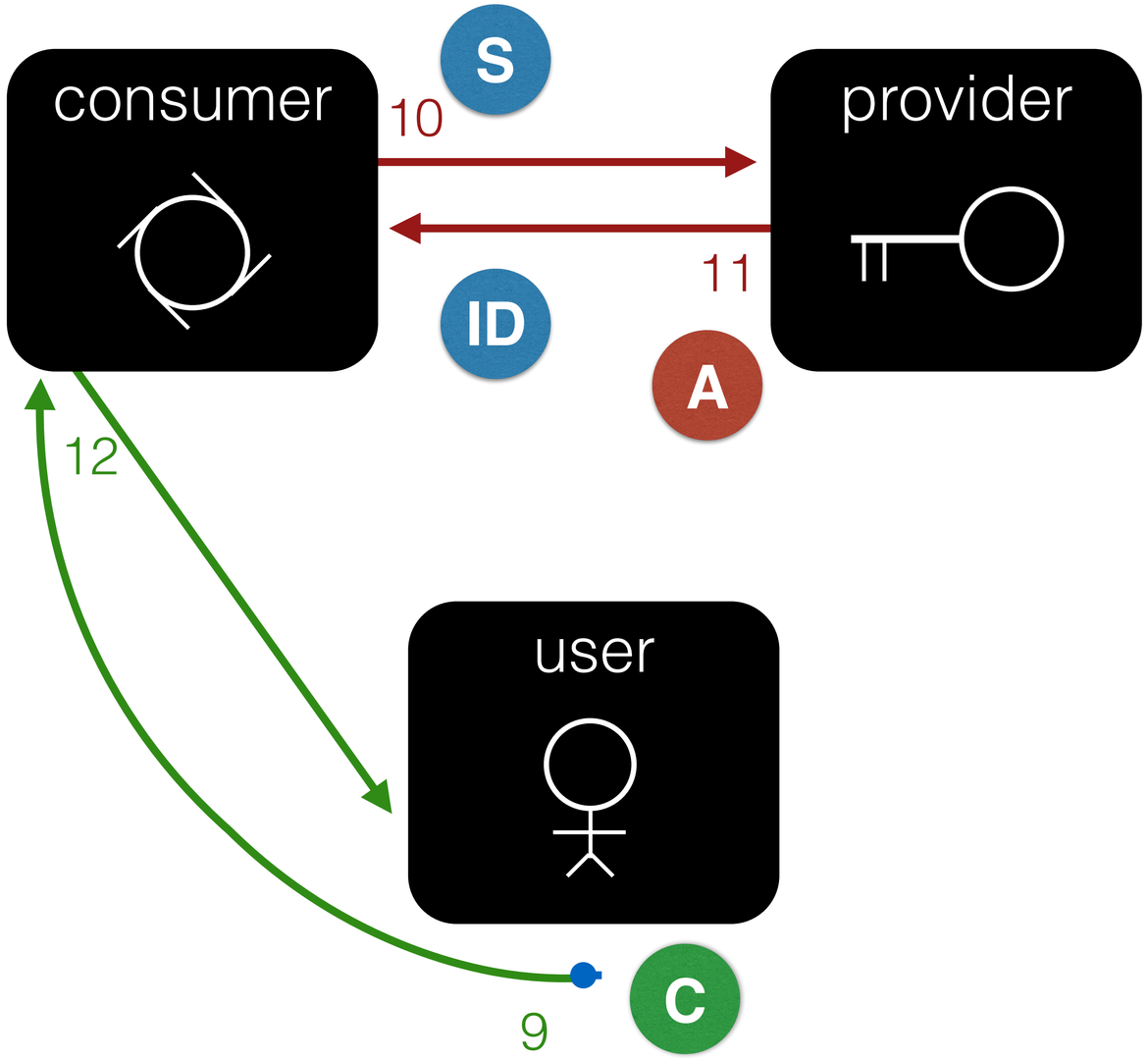


http://consumer.com/confirm?authcode=789

http://provider.com/oauth/authorize?client_id=123&callback_url=consumer.com/confirm&scope=read

http://provider.com/oauth/token?client_id=123&client_secret=911&authcode=789

OAUTH - APP AUTHENTICATION



9.Request: callback URL

10.Request: authorization

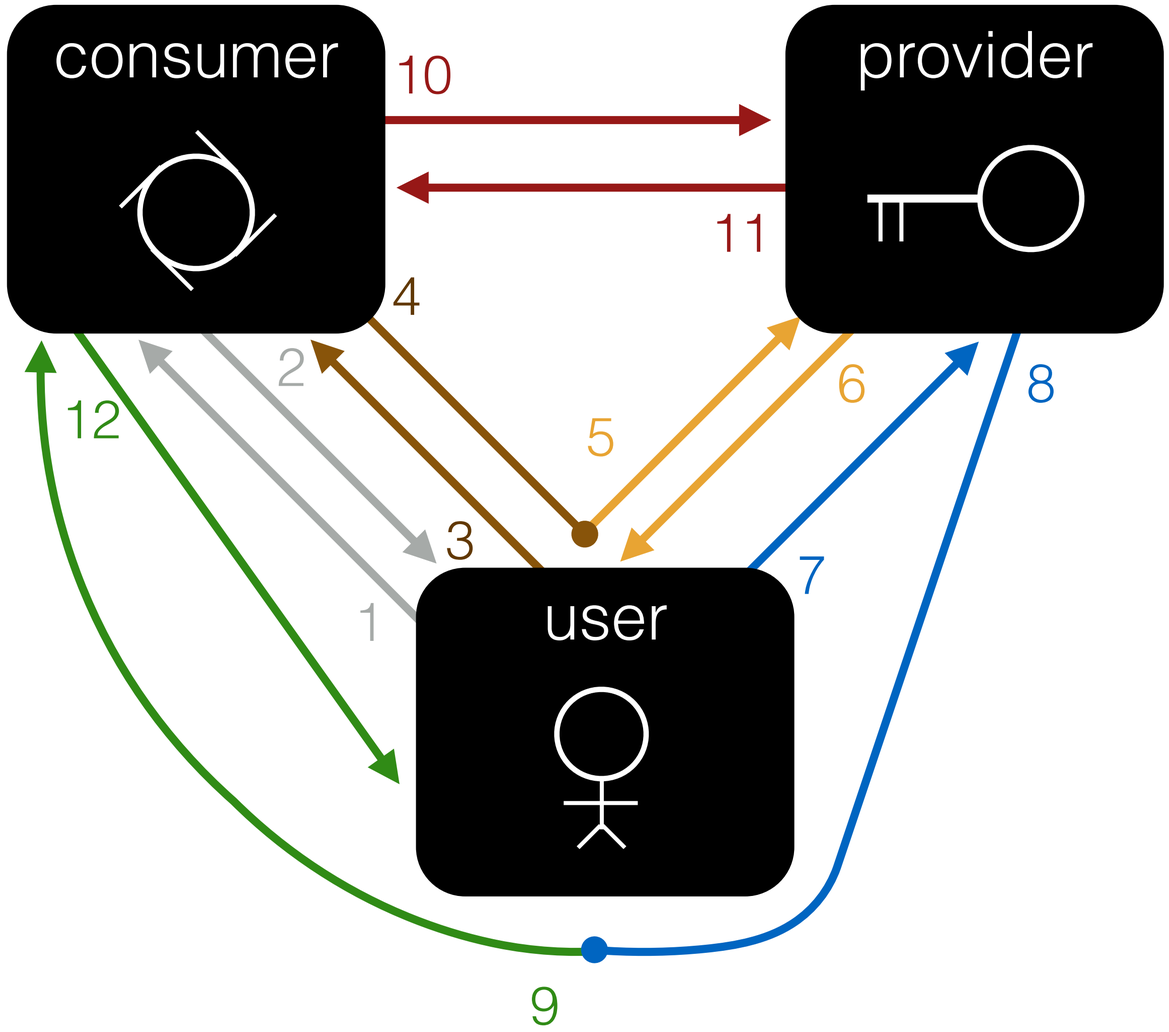
11.Response: access token

12.Response: yay!

http://consumer.com/confirm?authcode=789

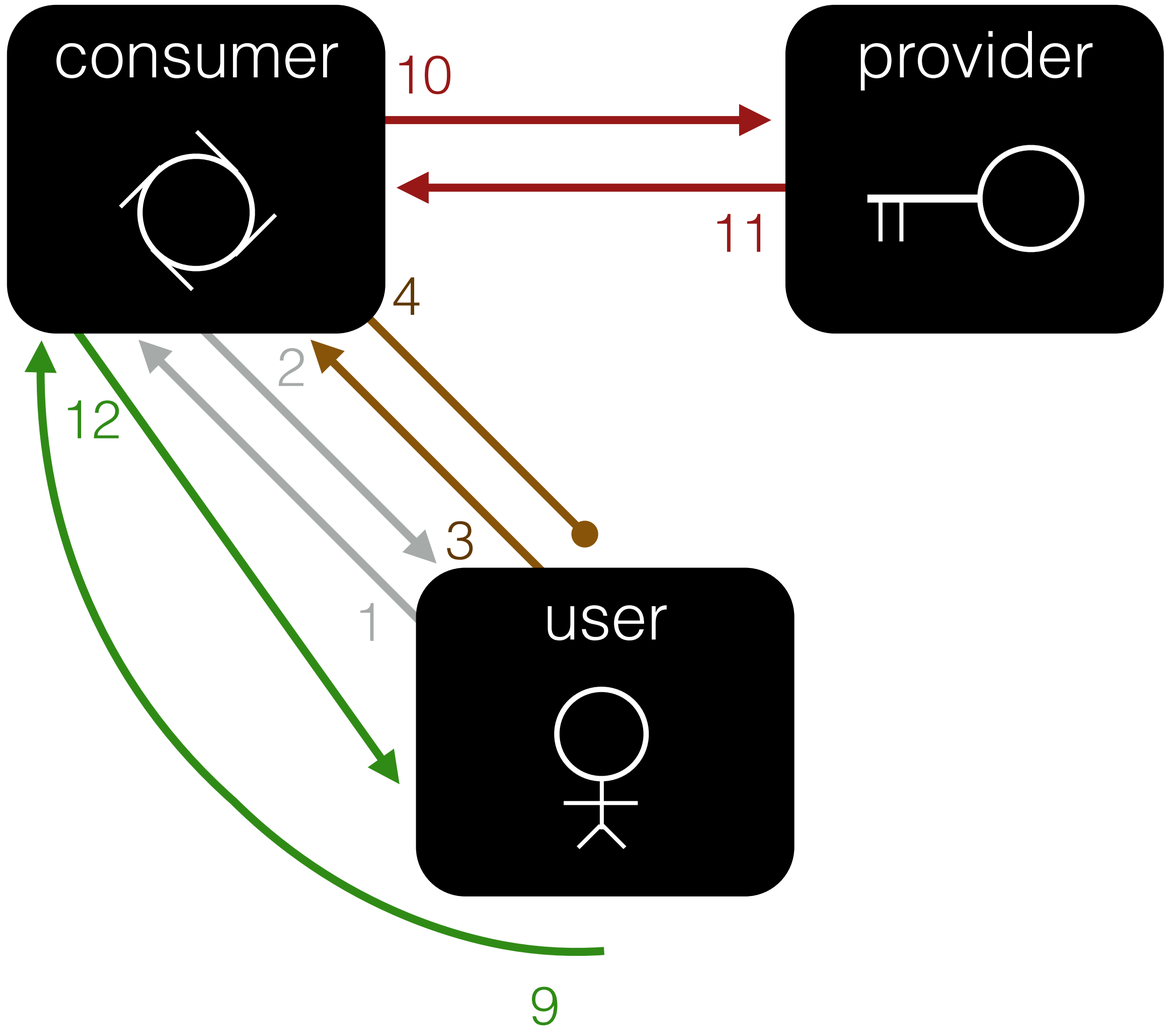
OAUTH - ALL OF IT

1. Request (user to app): load login page
2. Response (app to user): rendered login page
3. Request (user to app): allow app to use provider as me [user petitions app for a special contract to allow the app to do certain things on the user's behalf]
4. Response (app to user): redirect to provider login, passing along (to provider) an app id, a success "callback URL", and a permissions "contract" [app transfers this petition to provider]
5. Request (implicit, user to provider): load login page
6. Response (provider to user): rendered login page
7. Request (user to provider): login to provider [the user signs the contract]
8. Response (provider to user): on success, redirect to callback URL, passing along a new temporary code [the provider approves the user's signature]
9. Request (implicit, user to app): initiate callback
10. Request (app to provider): request for authorization given temporary code and app secret key [the app signs the contract]
11. Response (provider to app): on success, passes back an access token [the provider approves the app's signature and puts the contract into effect]
12. Response (app to user): we're good to go!



OAUTH - CONSUMER ROLE

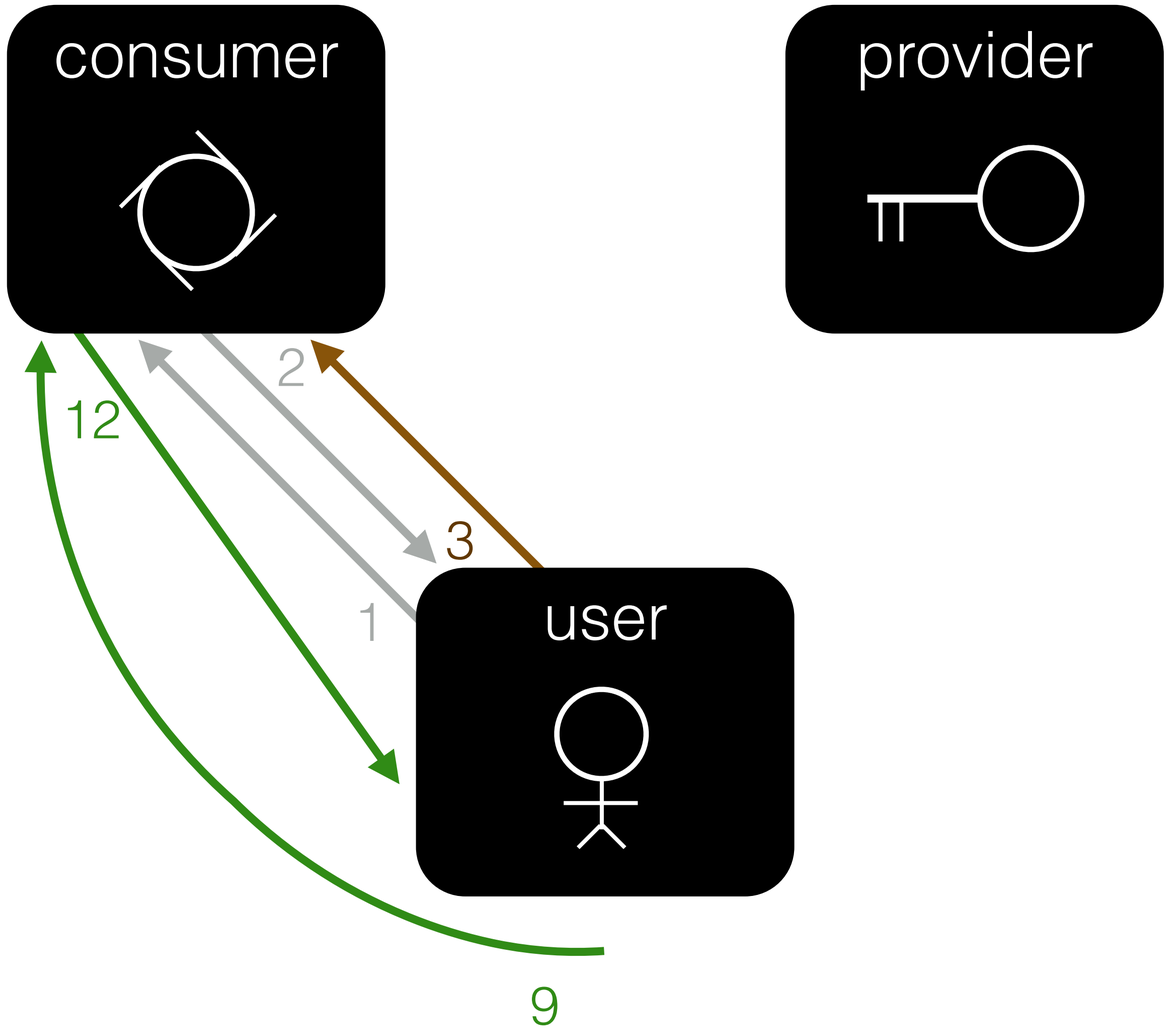
1.Request (user to app): load login page
2.Response (app to user): rendered login page
3.Request (user to app): allow app to use provider as me [user petitions app for a special contract to allow the app to do certain things on the user's behalf]
4.Response (app to user): redirect to provider login, passing along (to provider) an app id, a success "callback URL", and a permissions "contract" [app transfers this petition to provider]



9.Request (implicit, user to app): initiate callback
10.Request (app to provider): request for authorization given temporary code and app secret key [the app signs the contract]
11.Response (provider to app): on success, passes back an access token [the provider approves the app's signature and puts the contract into effect]
12.Response (app to user): we're good to go!

OAUTH - CONSUMER ROLE WITH PASSPORT

1.Request (user to app): load login page
2.Response (app to user): rendered login page
3.Request (user to app): allow app to use provider as me [user petitions app for a special contract to allow the app to do certain things on the user's behalf]



9.Request (implicit, user to app): initiate callback

12.Response (app to user): we're good to go!

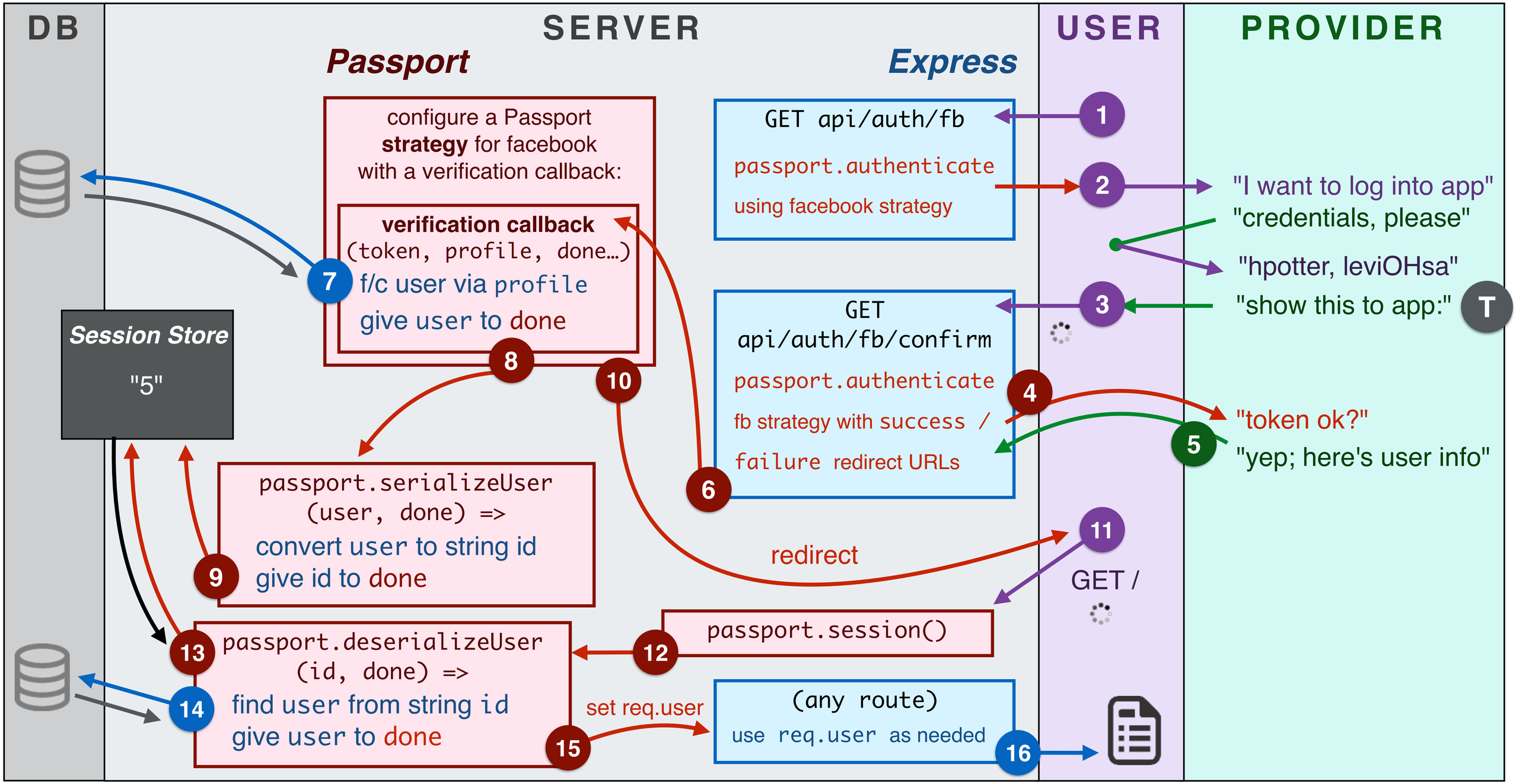

Passport

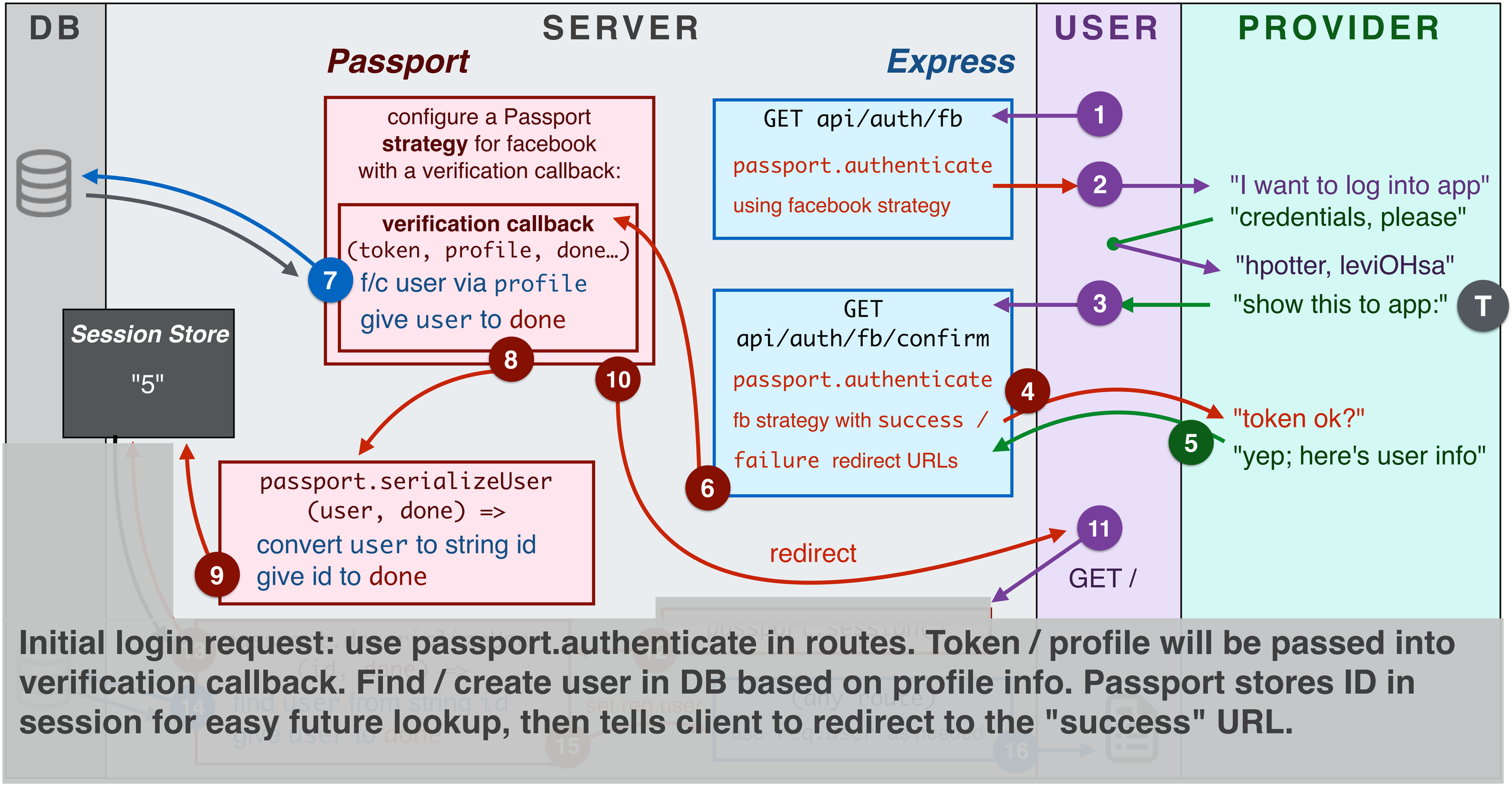
TL;DR

- Assuming you set everything up right:
- Client can log in by requesting GET /auth/fb or similar
- In routes you will now have req.user to check user info

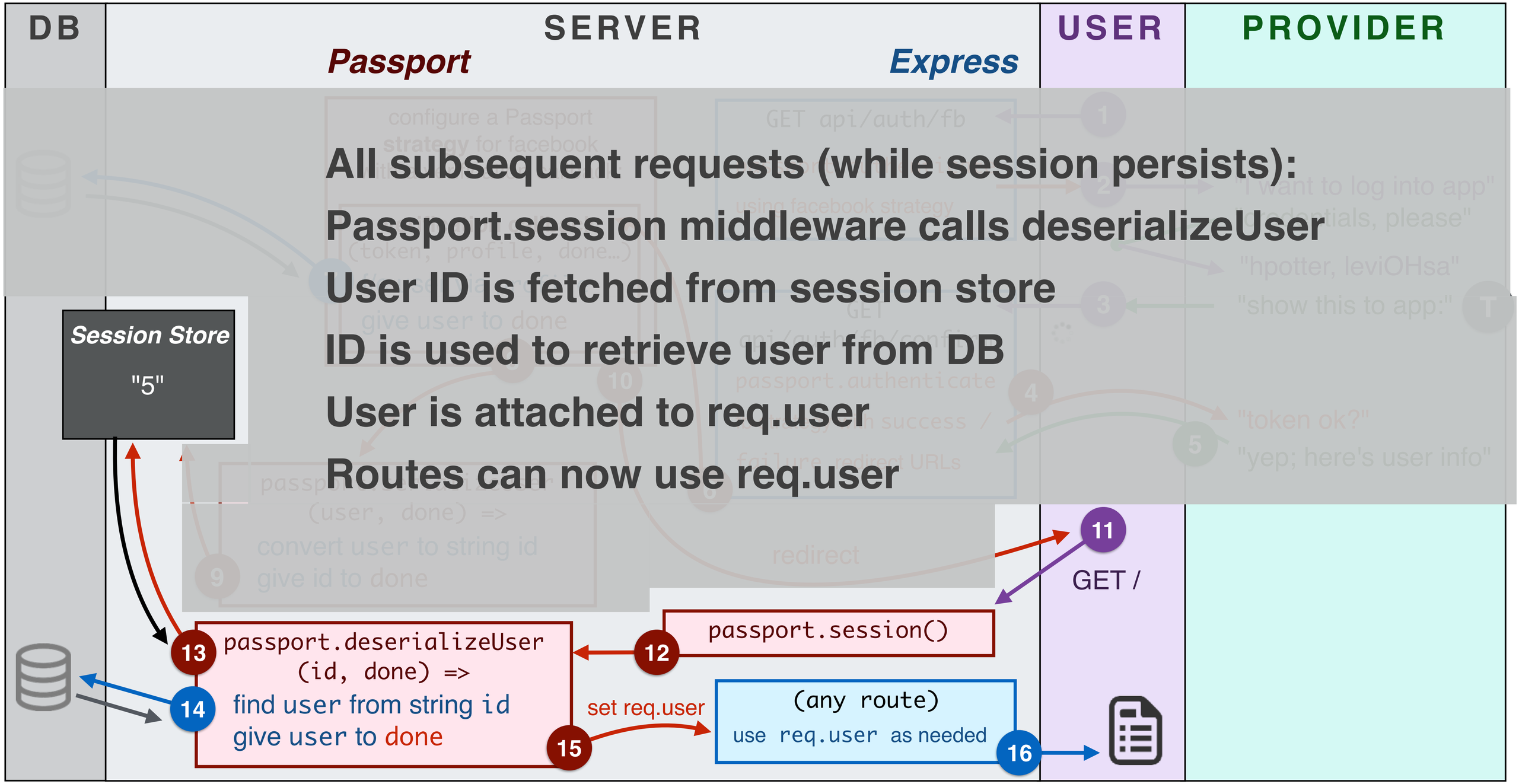
PASSPORT INGREDIENTS

- **attach passport.session() middleware**
  - Only after express-session
- **Define how to minimally store & look up user using session**
  - passport.serializeUser / passport.deserializeUser
- **Must configure a Strategy**
  - Strategy needs a *verification callback* you write — longest part
- **passport.authenticate (in two different routes)**
  - Uses strategy
  - Slightly different call in each route


DB
SERVER
USER
PROVIDER
Passport
Express
configure a Passport
strategy for facebook
with a verification callback:
verification callback
(token, profile, done…)
f/c user via profile
give user to done
GET api/auth/fb
passport.authenticate
using facebook strategy
"I want to log into app"
"credentials, please"
"hpotter, leviOHsa"
"show this to app:"
T
GET
api/auth/fb/confirm
passport.authenticate
fb strategy with success /
failure redirect URLs
"token ok?"
"yep; here's user info"
Session Store
"5"
passport.serializeUser
(user, done) =>
convert user to string id
give id to done
redirect
GET /
passport.session()
passport.deserializeUser
(id, done) =>
find user from string id
give user to done
set req.user
(any route)
use req.user as needed
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16



Initial login request: use passport.authenticate in routes. Token / profile will be passed into verification callback. Find / create user in DB based on profile info. Passport stores ID in session for easy future lookup, then tells client to redirect to the "success" URL.


DB
SERVER
Passport
Express
USER
PROVIDER
All subsequent requests (while session persists):
Passport.session middleware calls deserializeUser
User ID is fetched from session store
ID is used to retrieve user from DB
User is attached to req.user
Routes can now use req.user
Session Store
"5"
11
GET /
passport.session()
12
13
passport.deserializeUser
(id, done) =>
find user from string id
give user to done
14
set req.user
15
(any route)
use req.user as needed
16

AUTHER™