The Web Remembers You

- How does a shopping cart stay full?
- How do I log in?
- How does Google remember my past queries?

Session Requirements

- Goal: maintain state with stateless HTTP
- This is called a **session**

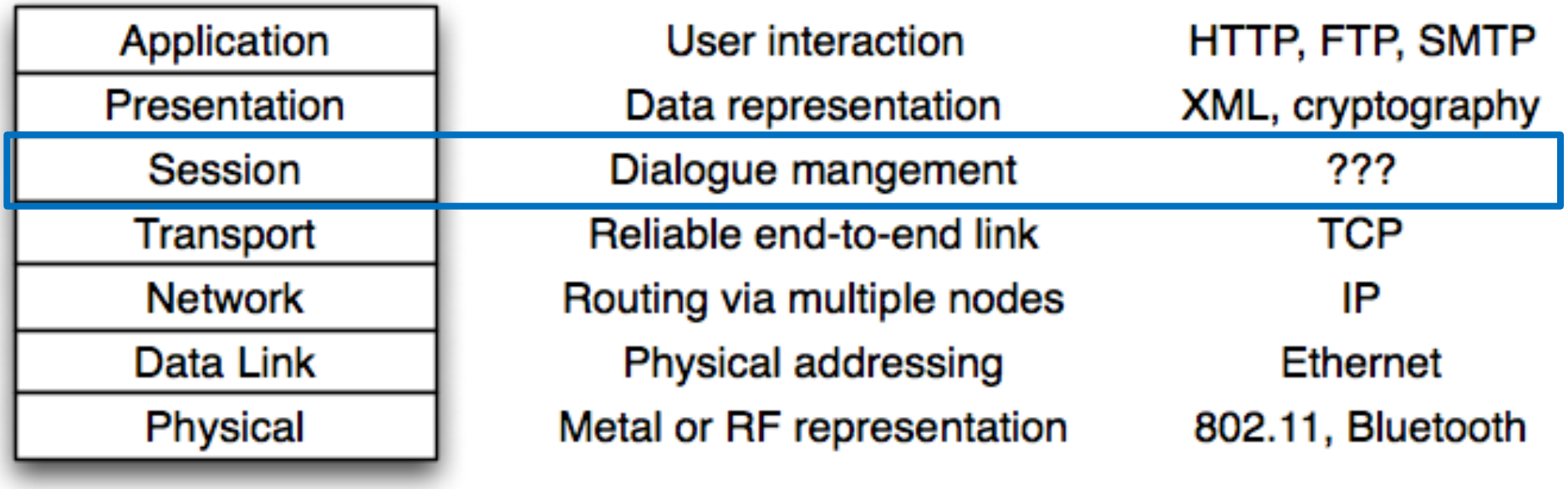
But HTTP is stateless!!!

- Principle of the web is to build in layers, with each layer as simple as possible.
  - We saw this in TCP over IP
- Could have built state into HTTP
  - State not always required
  - When state is needed, the reasons are various, and corresponding sizes vary
  - Supporting all this in HTTP would have made it bloated
- So build state on top of HTTP

Sessions

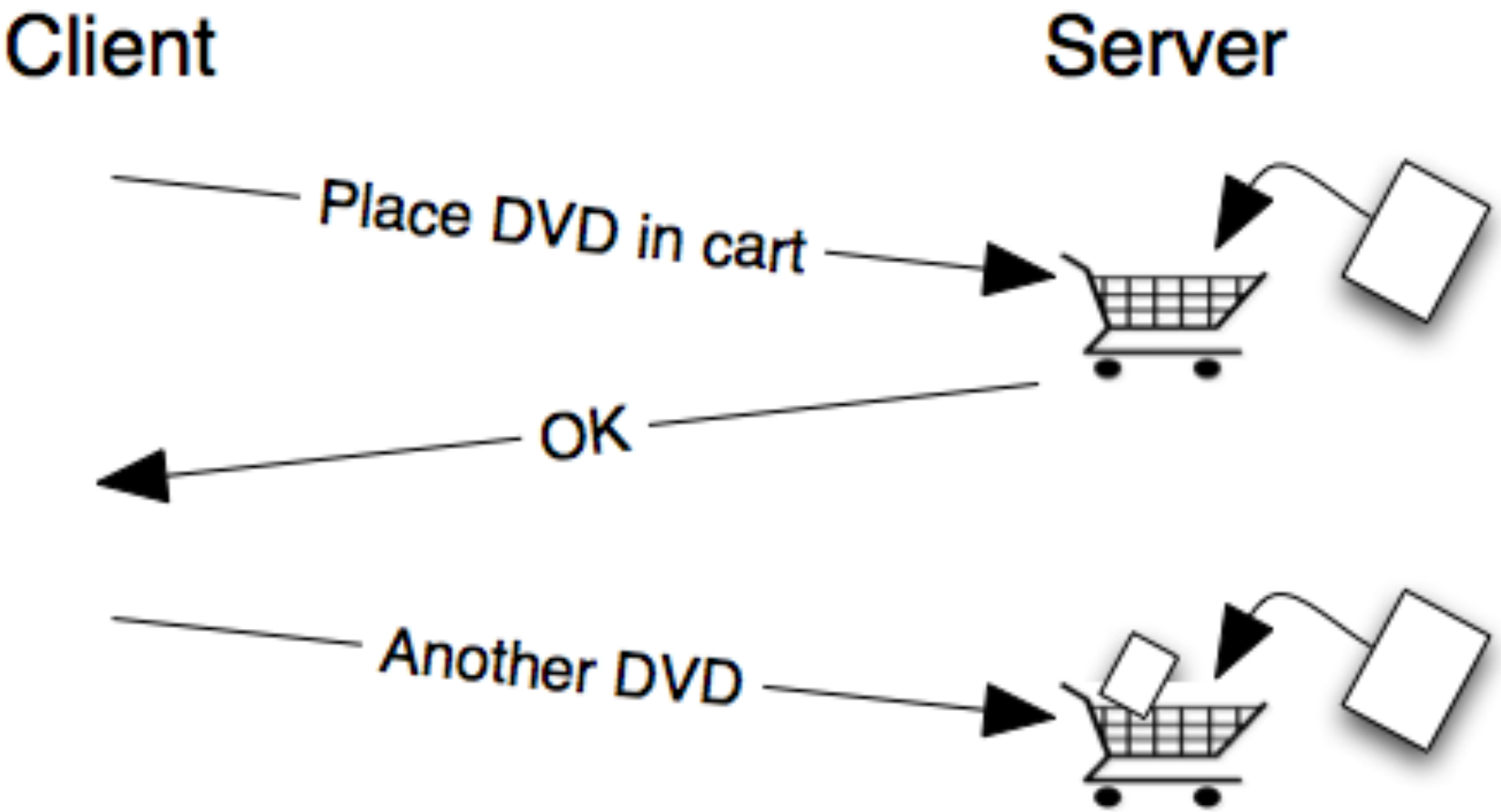
- A session is a single "interaction" between the site and user
  - Precise definition depends on application
  - HTTP is session-less
- TCP has connections, similar to sessions
  - Relationship between TCP and "sessions"?
  - HTTP 1.0 set up a fresh TCP connection for each request
  - HTTP 1.1 leaves TCP connection up for "some time"
    - How long is a performance optimization

Network Protocol Stack Model



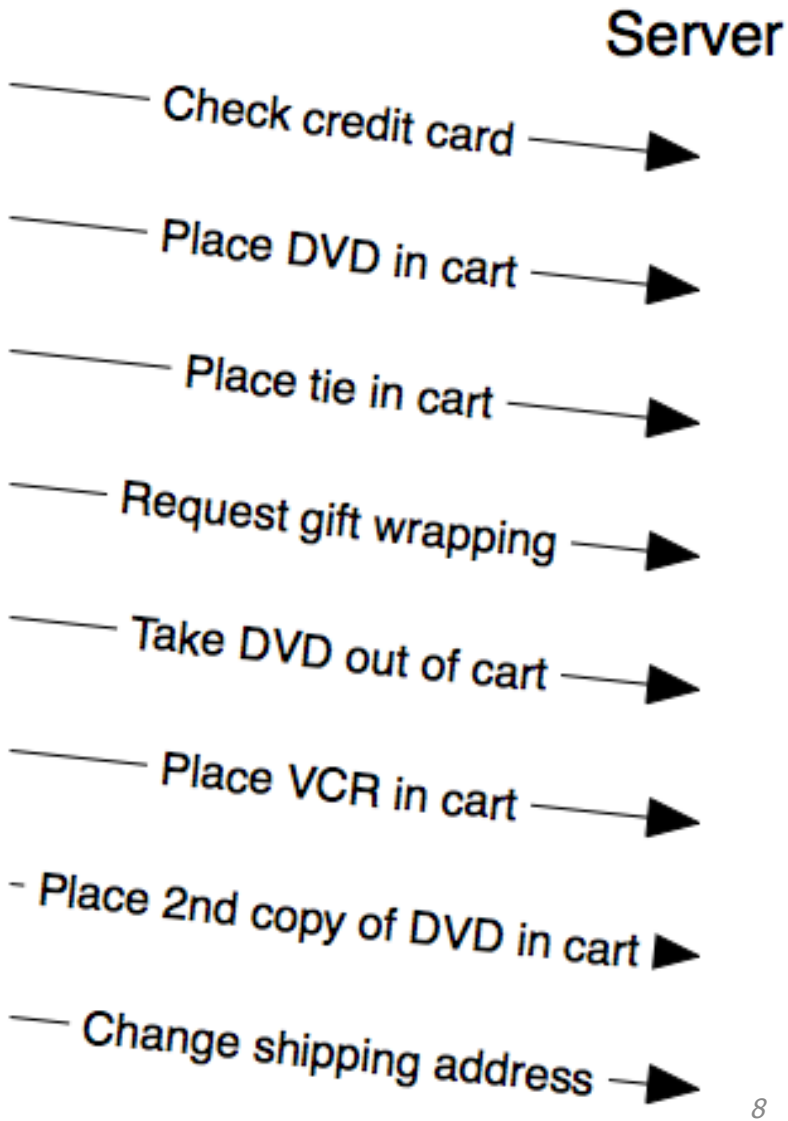
Application	User interaction	HTTP, FTP, SMTP
Presentation	Data representation	XML, cryptography
Session	Dialogue mangement	???
Transport	Reliable end-to-end link	TCP
Network	Routing via multiple nodes	IP
Data Link	Physical addressing	Ethernet
Physical	Metal or RF representation	802.11, Bluetooth

Session Perspectives - Client



Session Perspectives: Server

- A server must track many sessions at once



Stateful HTTP?

- HTTP is stateless, so we want to use a "session protocol" on top of it
  - Like TCP does on IP
- But, uh, there's no such thing as a "session protocol"
- Implemented at app-layer instead
  - State maintained in session variables
  - Data stored in one request can be accessed by later request, as long as within same session

Server Session Model

- Sessions like files, or call-stack frames
  - Collection of attr/value pairs
  - E.g., **`name=mike, numEmails=20`**
- Sessions explicitly opened, closed
  - Python/Flask does a lot for you by default
- Server issues:
  - When to create + close sessions?
  - How to link HTTP requests to a session?
  - How to link sessions to users?
  - How to store session data?

Server session processing

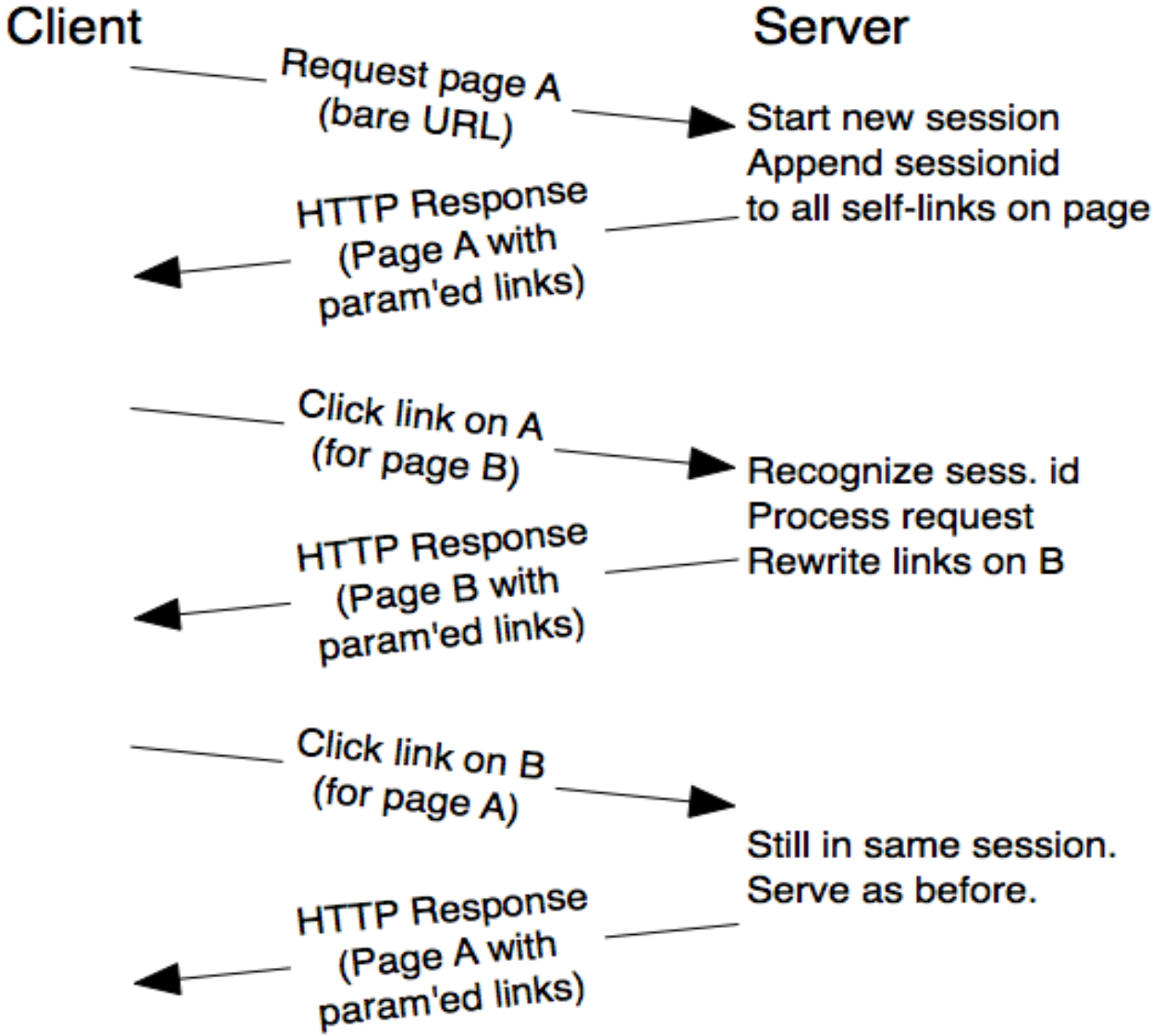
1. HTTP request arrives
   - If new session, init session structure
   - Else, load session structure from storage
2. Service HTTP Request; update session
   - If end of session, dealloc session struct
   - Else, save session updates to storage
3. HTTP response to client

Begin and end

- Starting a session is easy
  - Is request associated with a session?
- Ending is harder
  - We can't rely on logout
  - Timeouts needed for almost all apps
    - When should the online game be reset?
    - When should Google forget your search?
    - When has your cart been abandoned?
    - When have you started searching for a different flight?
  - Timeout from first request or most recent?

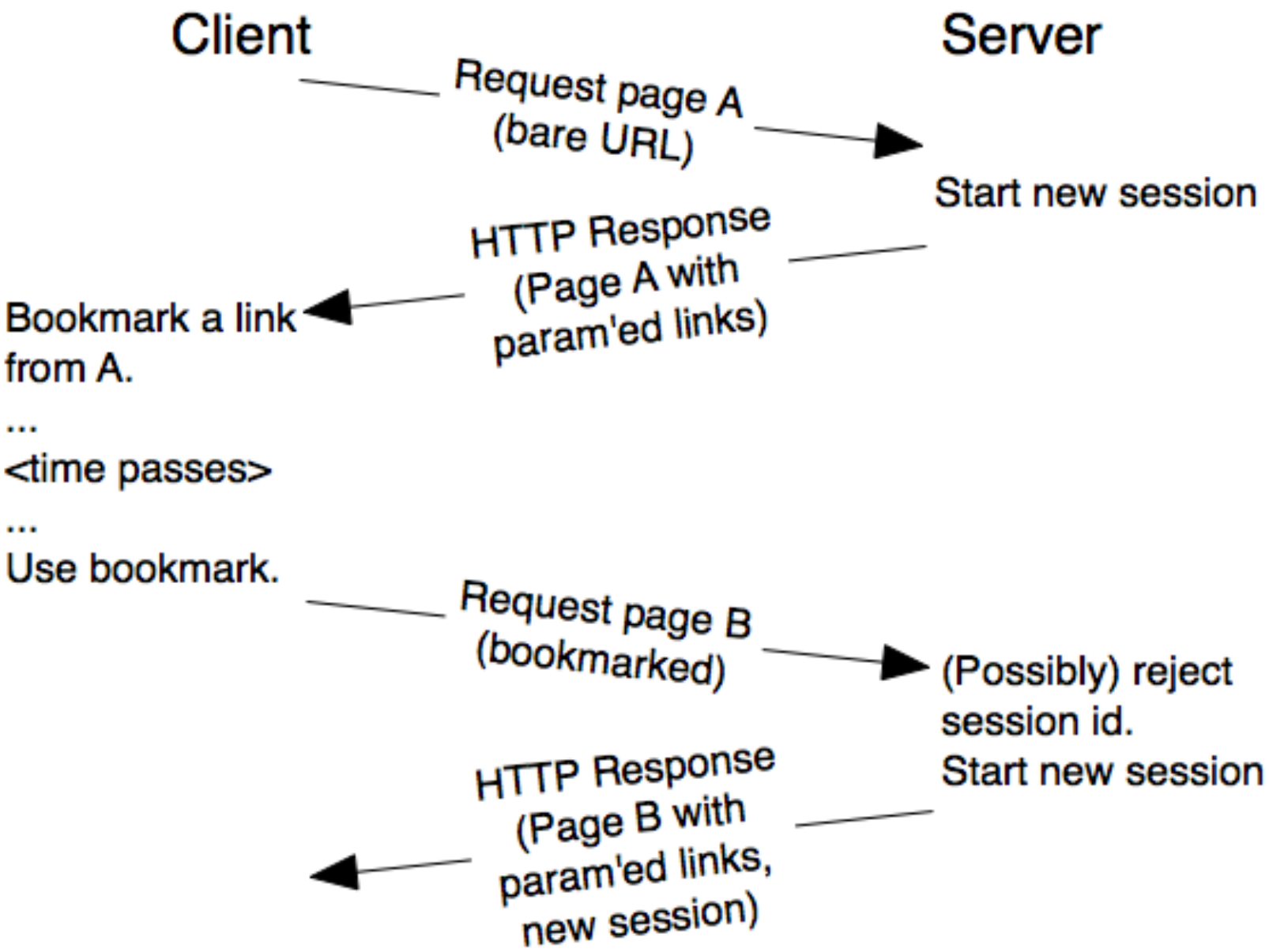
Request-to-Session Mapping

- One approach: URL Encoding
  - http://google.com/search?q=hello
  - URL parameters come after ?
  - Attr/val pairs
  - Rewrite the URLs on a page to reflect the session id or other embedded data


Client
Server
Request page A
(bare URL)
Start new session
Append sessionid
to all self-links on page
HTTP Response
(Page A with
param'ed links)
Click link on A
(for page B)
Recognize sess. id
Process request
Rewrite links on B
HTTP Response
(Page B with
param'ed links)
Click link on B
(for page A)
Still in same session.
Serve as before.
HTTP Response
(Page A with
param'ed links)

Bookmarking

- What if the session is no longer valid?
- What if “sessionized URLs” are bookmarked?
  - Shared?
  - Intercepted?

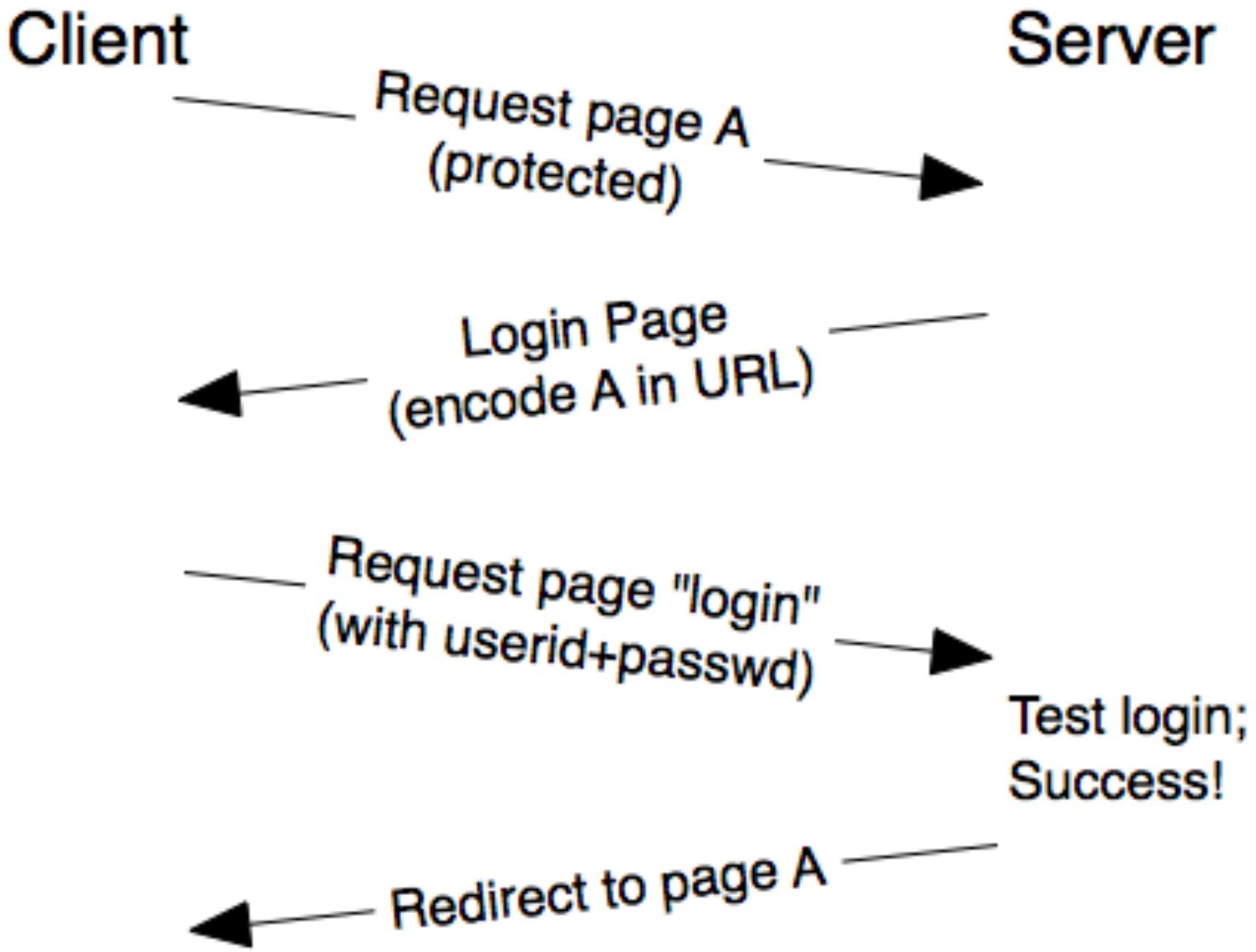

Client
Server
Request page A
(bare URL)
Start new session
HTTP Response
(Page A with
param'ed links)
Bookmark a link
from A.
...
<time passes>
...
Use bookmark.
Request page B
(bookmarked)
(Possibly) reject
session id.
Start new session
HTTP Response
(Page B with
param'ed links,
new session)

Bookmarking

- What if the session is no longer valid?
- What if URLs are bookmarked?
  - Shared?
  - Intercepted?
- Session ids should:
  - Have some resistance to guess-attacks
  - Be unlikely to be accidentally replicated
    - Incrementing a counter is bad
  - Contain validating data (like timestamp)

Logins

- Sessions can be anonymous
- Sometimes nicer to authenticate
  - Store state per-user basis, not per-session
  - User can move to different machines
  - Some state should be long-lasting
- Encode authenticated user id
  - Often helpful to encode userid *and* session
  - Authentication next week
- Combine login w/webpage access ctrl


Client
Server
Request page A
(protected)
Login Page
(encode A in URL)
Request page "login"
(with userid+passwd)
Test login;
Success!
Redirect to page A

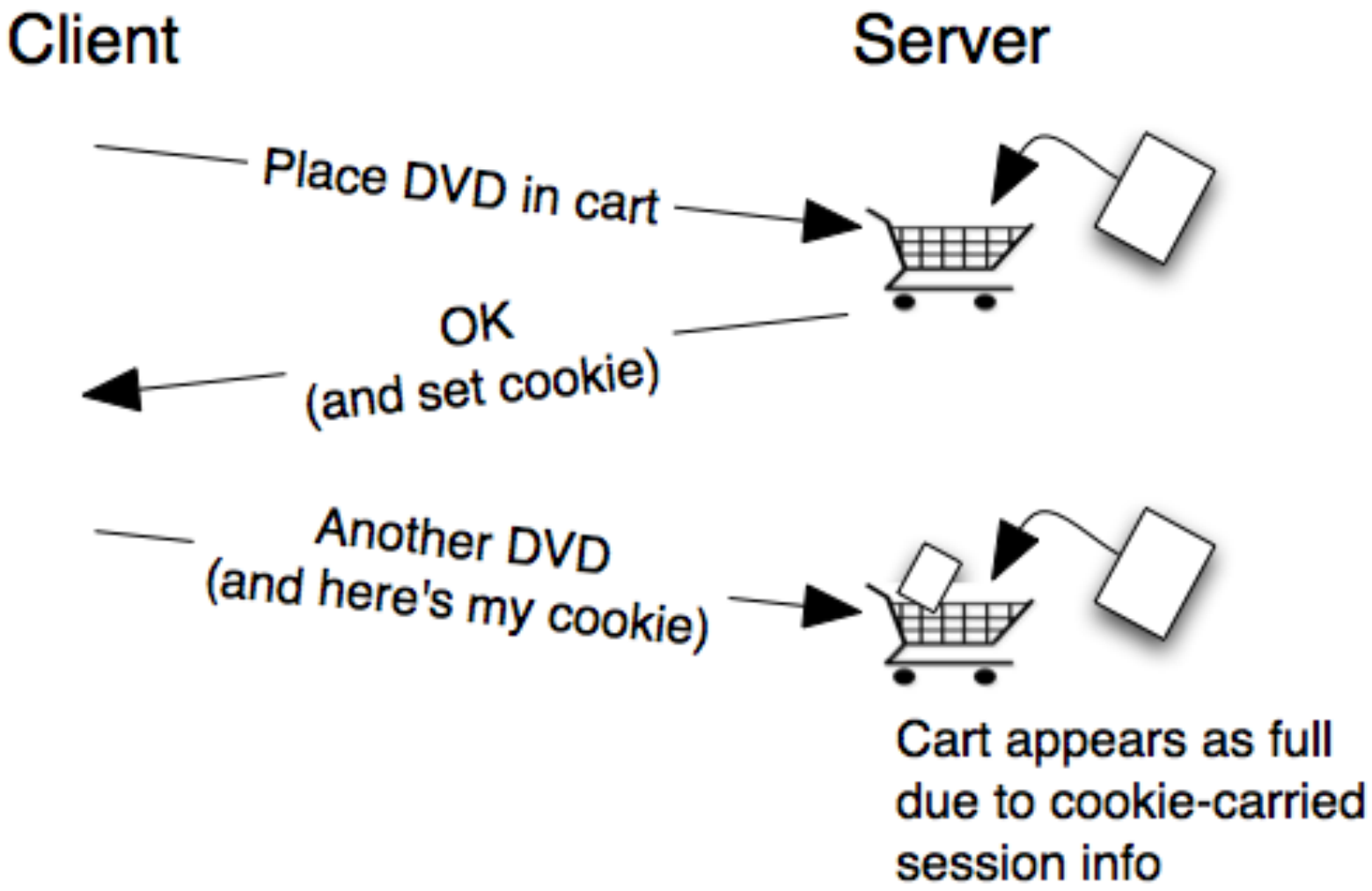
Sessions and Usability

- Login failure has default 401 response
  - Nicer to redirect to login page
- If request requires login first, don't just dead-end at login page
  - Store target as session var; visit after login
- Store userid for faster future login

Cookies

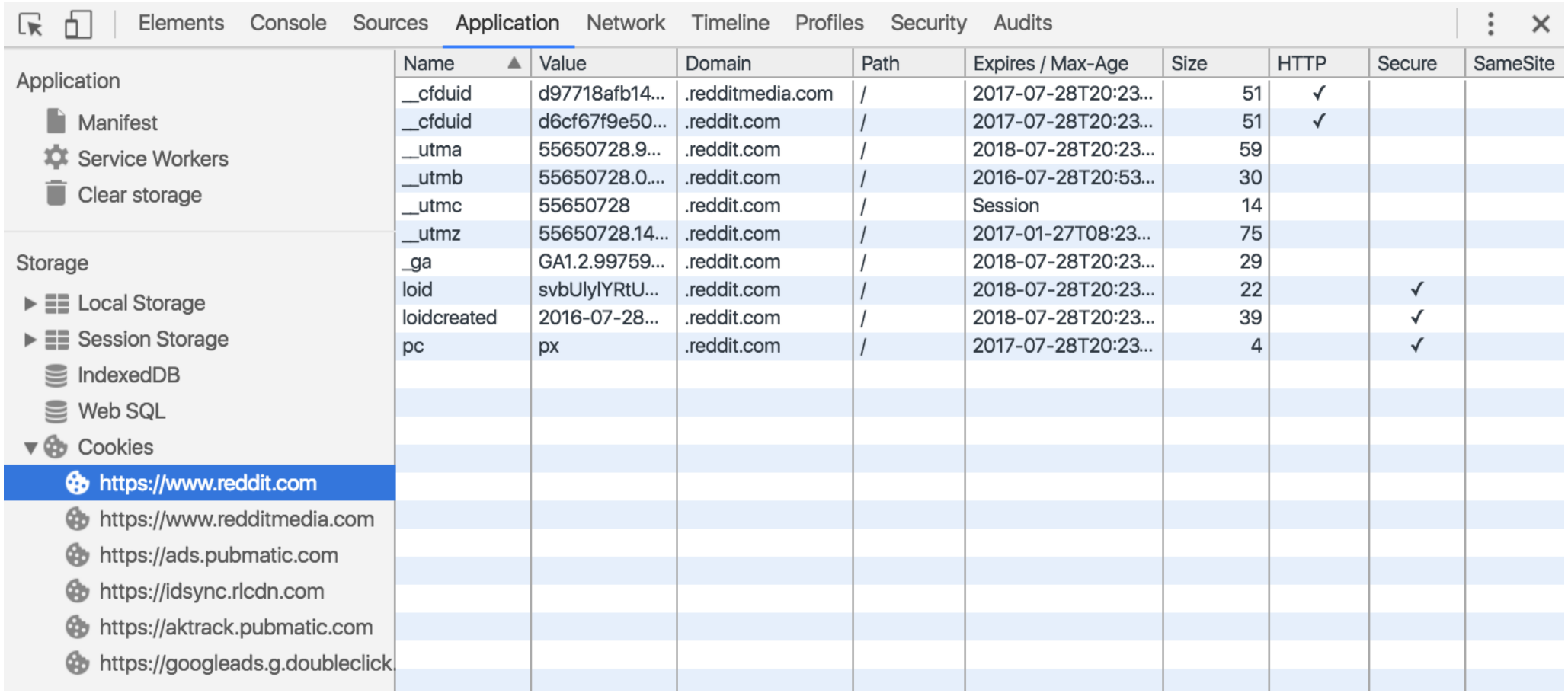
- URL-encoding is not perfect
  - What if you type in the URL yourself?
  - Some software refuse too-long URLs
  - Ugly
- Cookies are small files on client machine
  - Carry state between HTTP requests
  - attr/val pairs, just like URL params
- Set by server, but not requested
- Sent by client, but never edited
- Either side can delete/ignore cookies

Cookies


Client
Server
Place DVD in cart
OK
(and set cookie)
Another DVD
(and here's my cookie)
Cart appears as full
due to cookie-carried
session info

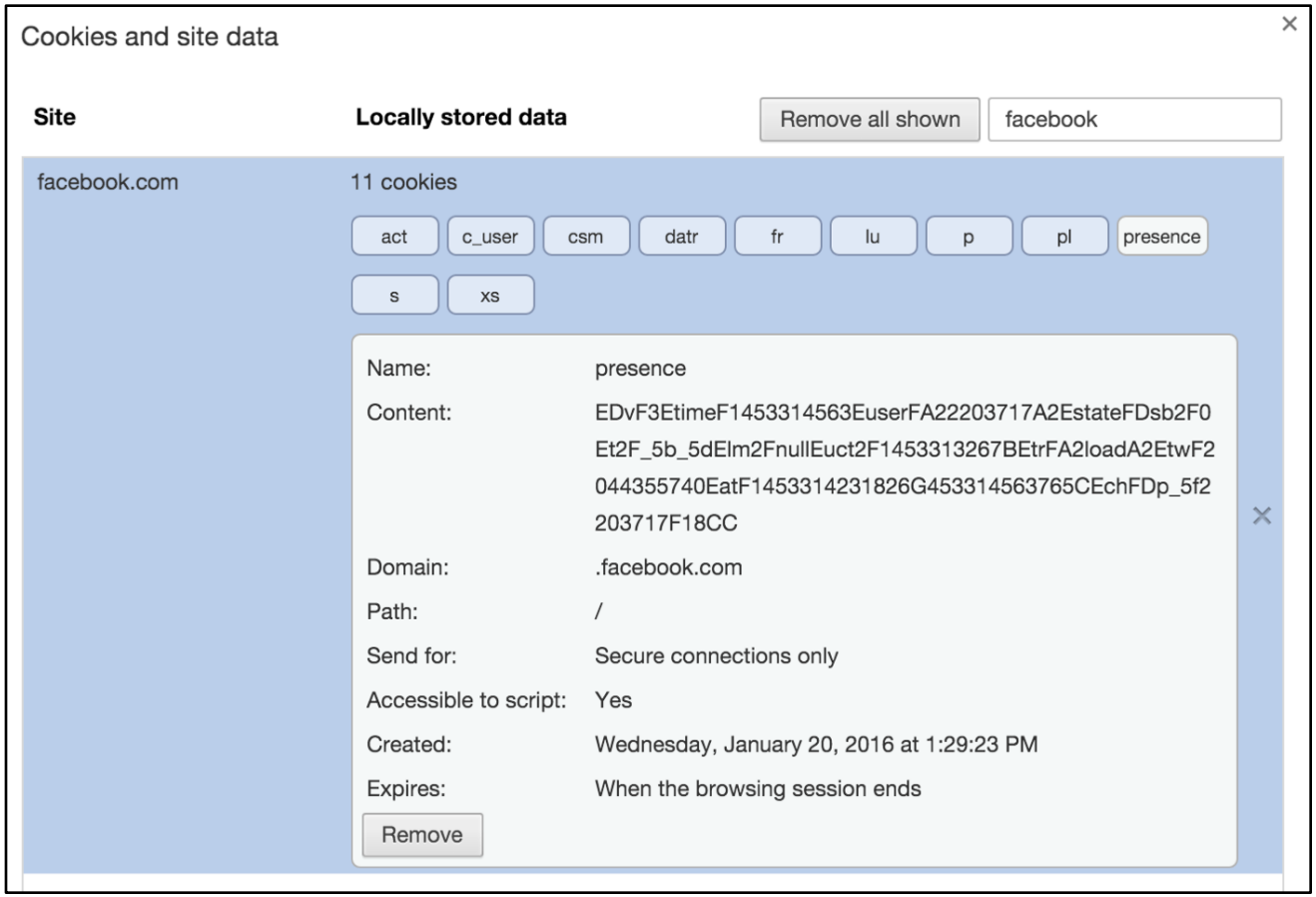
Example: cookies in Chrome

- Menu -> Tools -> Developer Tools



Example: cookies in Chrome

- **Navigate to** `chrome://settings/cookies`

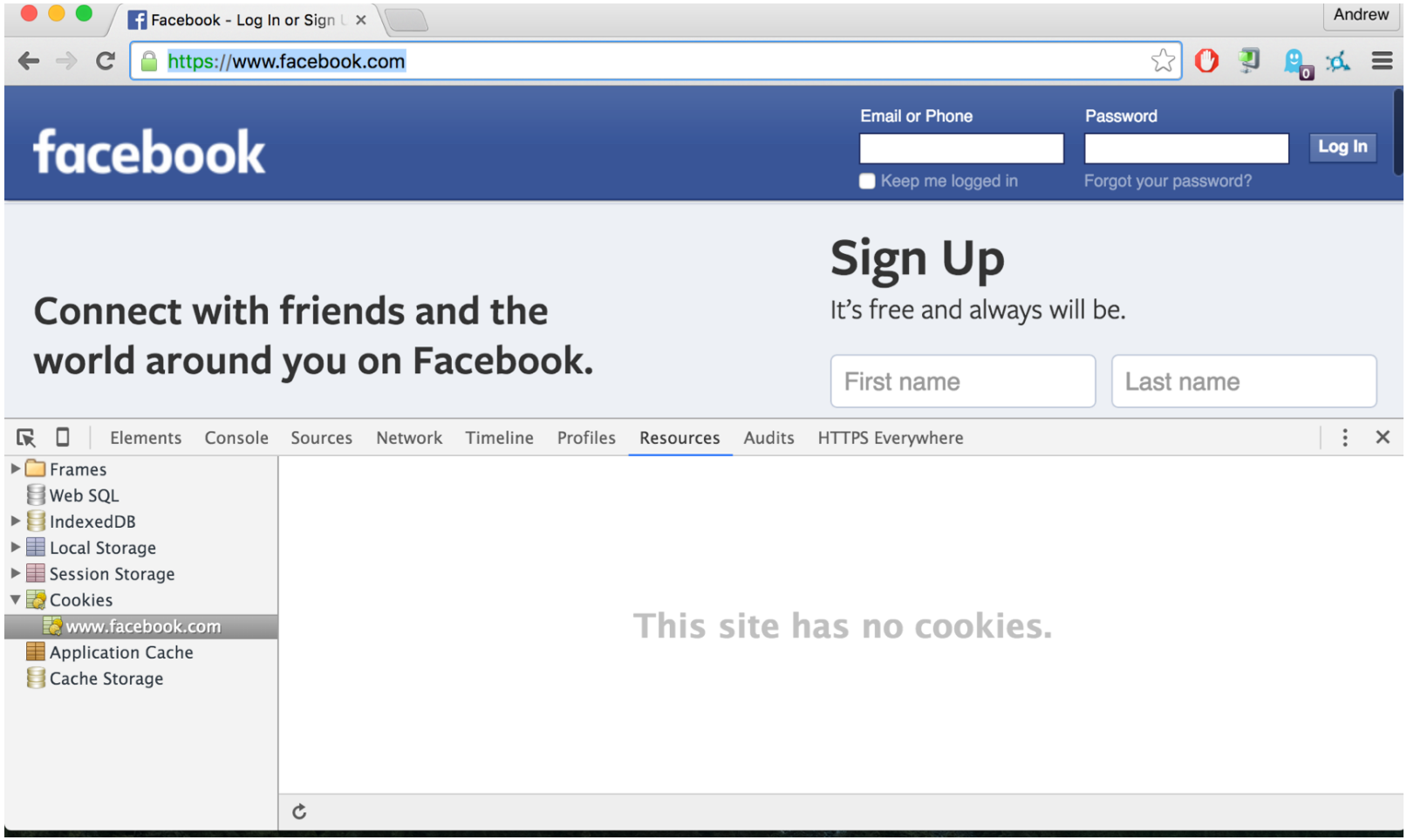


Cookie Contents

- **Name** is up to the server
- **Content** is up to server: encrypted? OK!
- **Domain** used by browser per-domain, total limits
- **Path** specifies scope of cookie
  - catalog vs catalog/page1.html
- **Expiration** tells client when to delete
- **Secure** is how cookie may be transmitted
- Supported by HTTP
- Cookies only over-writable by src domain and path-prefix

Example: cookies in Chrome

- Try this: delete all the cookies
- You're no longer logged in!



Uses and Abuses

- Cookies make sessions quiet, ubiquitous
- Cookies also add to HTTP overhead
- Best practice is to put as little info in a cookie as possible. Keep bulk on server.

- Bad example: keeping application state (like actual cart contents) in cookie
- Cookies could arrive in different orders, creating concurrency problems!
  - Known issue with Python/Flask + AJAX

Uses and Abuses

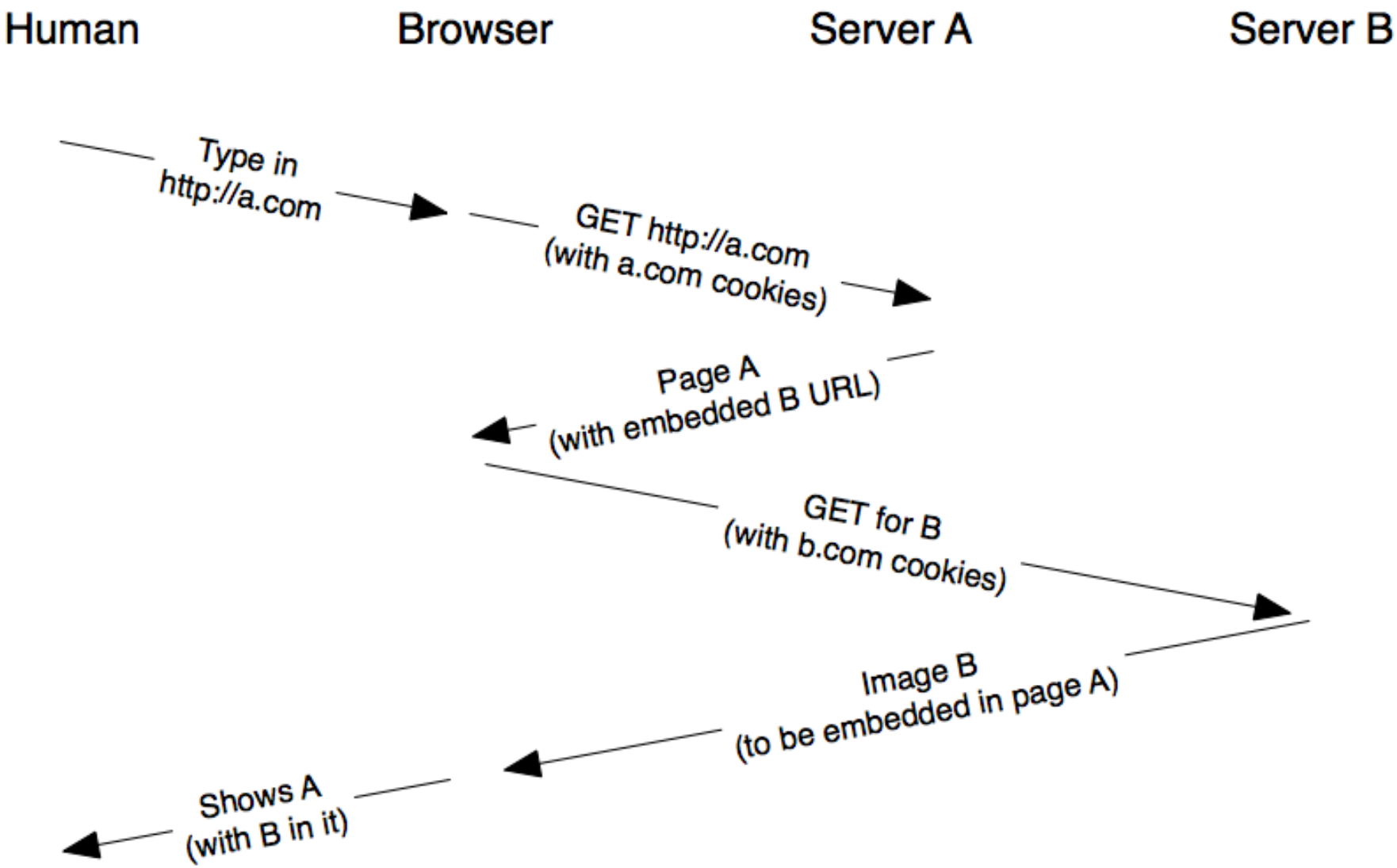
- Third-party cookies
- Page may contain objects from many srcs
  - Images, ads, other plugins
- These "3rd-party" objects set/get cookies
- AdSense can track across sites this way
- "Web Bugs" are invisible images or frames, used for tracking

Third-party Cookies

- Example: the Google "+1" and Facebook "like" buttons embedded in other websites can be used to track your activity on other pages

```
<script src="https://apis.google.com/js/platform.js"
  async defer></script>
<g:plusone></g:plusone>
```

Third-party Cookies



Who Uses Third-party Cookies?

- Companies that sell ads directly
  - Google and Facebook (obviously)
- Companies that sell information about you
  - Acxiom "one of the biggest companies you've never heard of" ($1B+)

- If you're not paying for the product, then you *are* the product
  - That's mostly how the business side of the web is structured

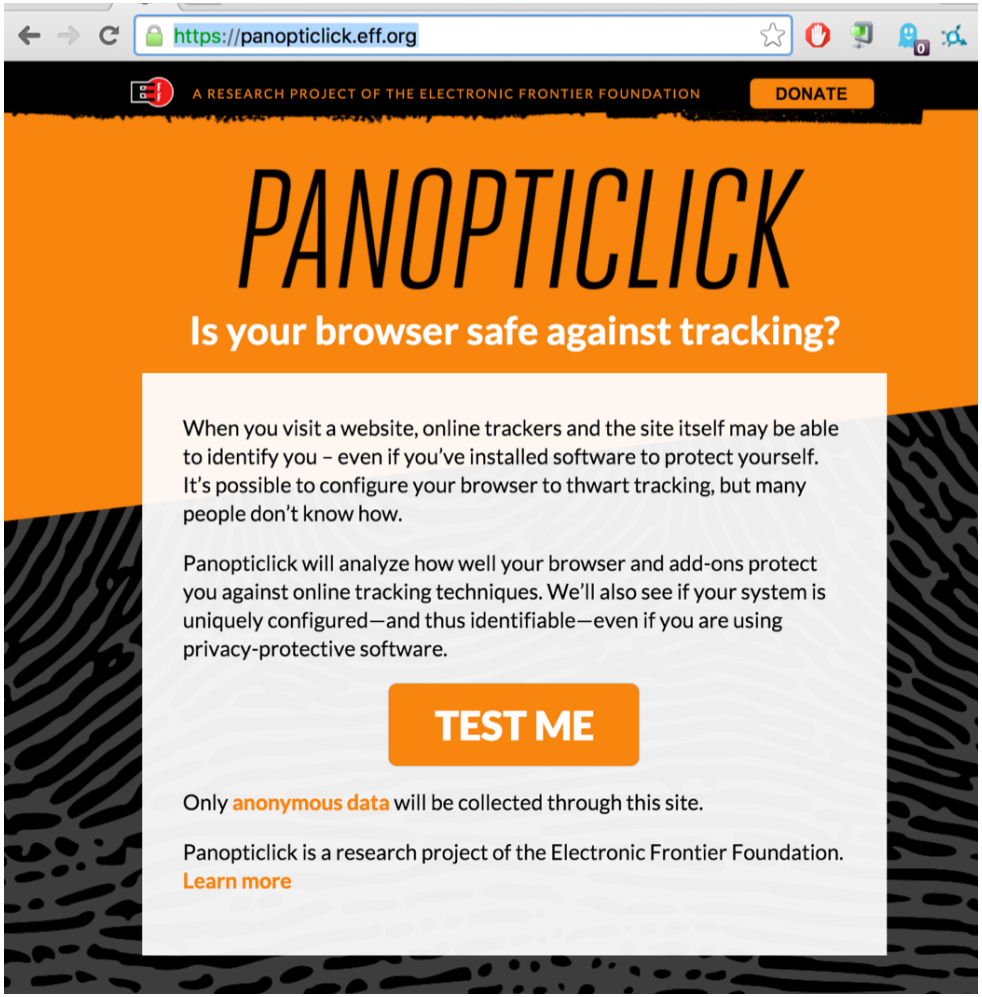
What Does Acxiom Have on Me?

- You can view part of the profile Acxiom has on you
- ....

- But to see it, you need to give them your name, address, email, social security number ...

Checking What You Send to Trackers

- https://panopticlick.eff.org/



Avoiding Trackers

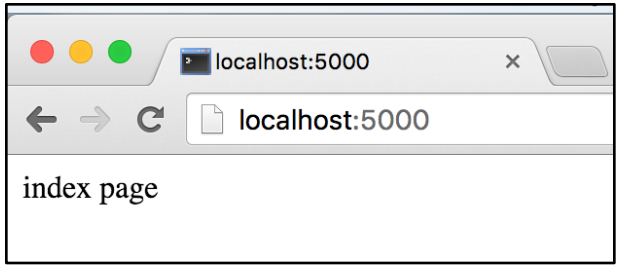
- Add-ons like Ghostry or ScriptNo block tracking ads
- Monitors embedded links and blacklists trackers

Discussion

- Many web companies make their money from information about users
- In exchange, they give you a "free" service (email, web search, a platform for gossip, etc.)
- Is it OK to block trackers?

Python/Flask Sessions How-to

- This will be helpful for project 2
- Start with no sessions

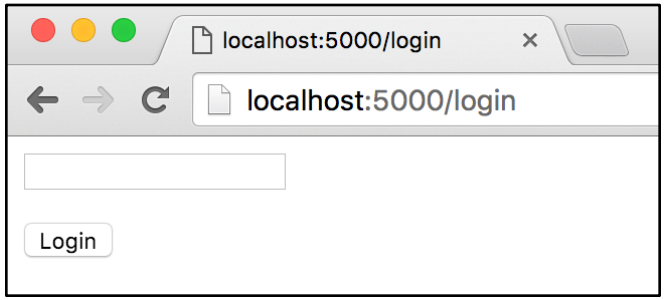



```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
  return 'index page'

if __name__ == '__main__':
  app.run()
```

Python/Flask Sessions How-to

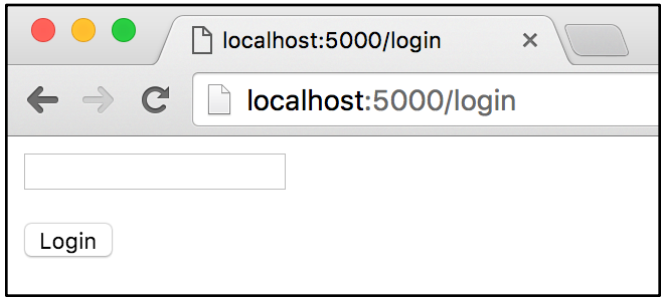


- Add a login page

```
# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    return '''
        <form action="" method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>'''
# ...
```

Python/Flask Sessions How-to



- Handle post request

```
# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
  if request.method == 'POST':
    print 'DEBUG ' + request.form['username']
    return redirect(url_for('index'))
  return '''
    <form action="" method="post">
      <p><input type=text name=username>
      <p><input type=submit value=Login>
    </form>'''
# ...
```

Python/Flask Sessions How-to

- Session object gives access to cookie contents and automatically gets/sets it on client machine

```
from flask import session

@app.route('/login', methods=['GET', 'POST'])
def login():
  if request.method == 'POST':
    print 'DEBUG ' + request.form['username']
    return redirect(url_for('index'))
  return '''
    <form action="" method="post">
      <p><input type=text name=username>
      <p><input type=submit value=Login>
    </form>'''
```

Python/Flask Sessions How-to

- Session cookies are encrypted on client's machine so they can't read it. Need a key for this.

```
from flask import session
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
@app.route('/login', methods=['GET', 'POST'])
def login():
  if request.method == 'POST':
    print 'DEBUG ' + request.form['username']
    return redirect(url_for('index'))
  return '''
    <form action="" method="post">
      <p><input type=text name=username>
      <p><input type=submit value=Login>
    </form>'''
```

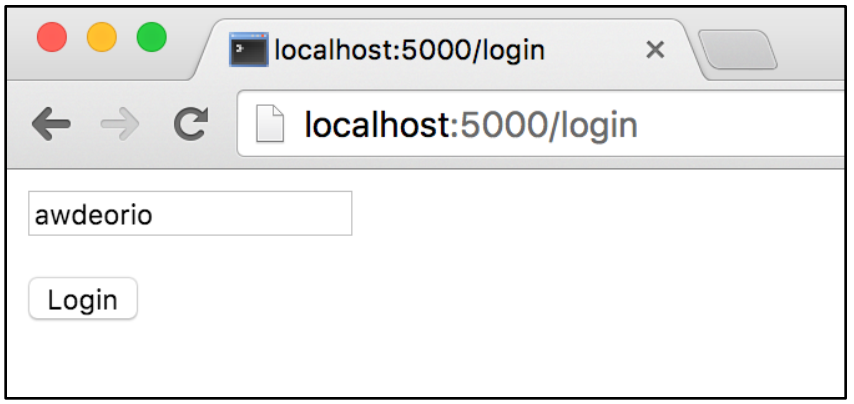
Python/Flask Sessions How-to

- Add new information to the cookie which will be automatically encrypted and stored on client host

```
from flask import session
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
@app.route('/login', methods=['GET', 'POST'])
def login():
  if request.method == 'POST':
    session['username'] = request.form['username']
    return redirect(url_for('index'))
  return '''
    <form action="" method="post">
      <p><input type=text name=username>
      <p><input type=submit value=Login>
    </form>'''
```

Python/Flask Sessions How-to

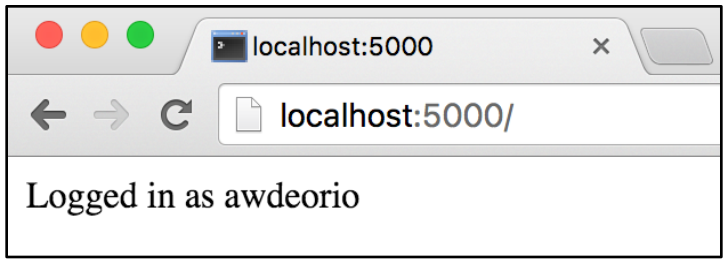
- Now you can log in



- And see the cookie

Python/Flask Sessions How-to

- Modify the main page to read contents of cookie and announce logged in user




```
@app.route('/')
def index():
    if 'username' in session:
        return 'Logged in as %s' %
          escape(session['username'])
    return 'You are not logged in'
```

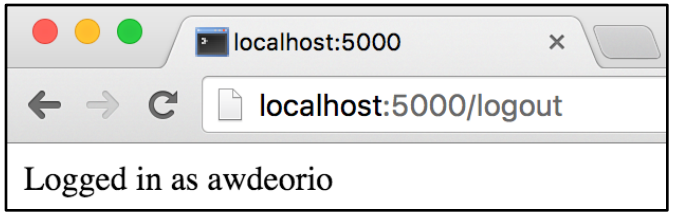
Python/Flask Sessions How-to

- Add a logout route which removes contents of cookie

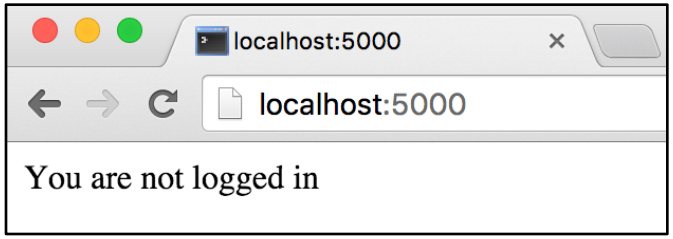
```
@app.route('/logout')
def logout():
  session.clear()
  return redirect(url_for('index'))
```

Python/Flask Sessions How-to

- Browse to logout URL



- And be redirected to main page, where the cookie no longer contains your login information



Debugging Sessions

- PostMan extension for Chrome
- Send requests manually