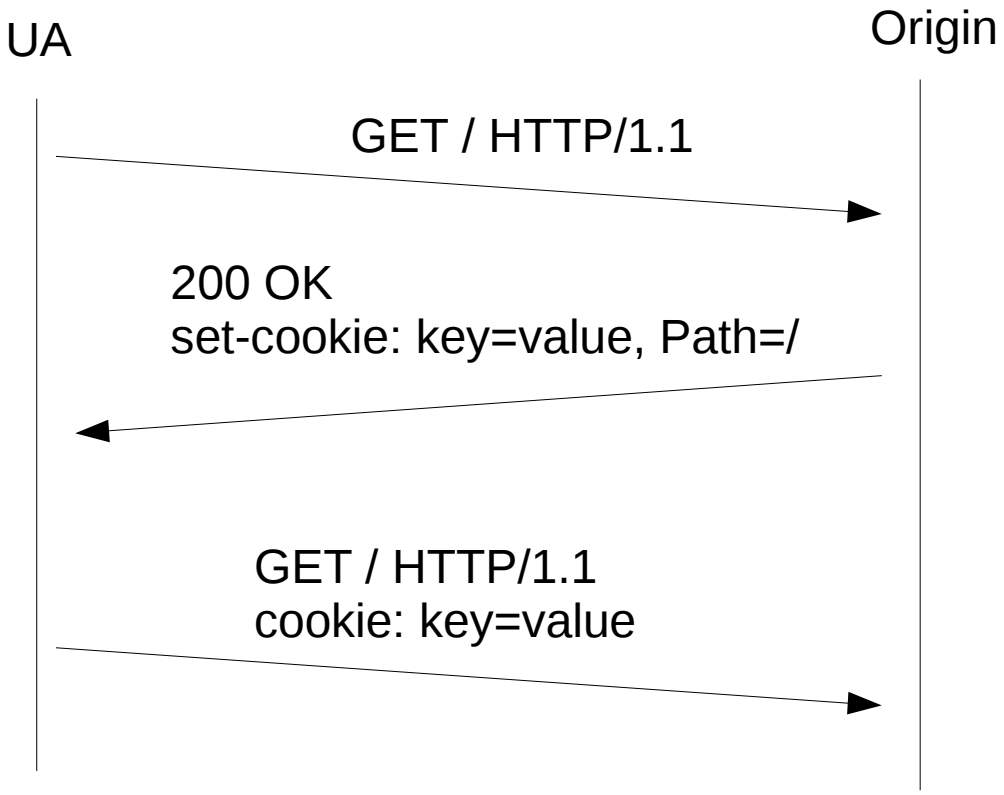
Http sessions

Contents

- Intro on state
- Http cookies
- Sessions
  - Client-side sessions
  - Server-side sessions

State via cookies

- Http is ***stateless***
  - No information is maintained between two requests
- Management of the state is added via ***Cookies***
  - ***set-cookie*** *response header and* ***cookie*** *request header*



Cookies

- **Key-value** data
  - *e.g., name=alice*
- **Additional fields**:
  - ***Domain validity***
    - *e.g., 'Location=/'*
  - ***Time Validity***
    - *Expiry time*
    - *Permanent*
- *Note on overhead:*
  - Data are transferred within each **request**
  - Usually, Web servers have ***maximum allowed size***

Cookies in Flask

- http://flask.pocoo.org/docs/1.0/quickstart/#cookies
- Get cookies
  - Read the dict-like cookies object in the request

```
request.cookies
```

- Write cookies
  - Manually write the set-cookie header in the response
  - Use the set_cookie method of the response object

```
r = make_response()
r.set_cookie(key, value)
```

- Unset cookies → set a cookie in the past

```
r.set_cookie(key, expires=0)
```

Cookies Flask Exercise

- Use the *sessions_flask* application as a basis
  - Implement the set_cookie handler
    - to return, if available, the current cookie values
    - to accept GET query parameters and set them in the cookies

Sessions

- A session is an **abstract concept**
- The server wants to **keep track** of requests from the **same client**

- Why?
  - Functionality
    - e.g., keep track of a form compiled over multiple
  - Targeted advertising
    - e.g., third party cookies
  - Authentication
    - e.g., alternative to ***Basic authentication***

Sessions and cookies

- Sessions require state → HTTP state is implemented via cookies

- ***Client-side session***
  - *Data are stored **in** the cookies*
    - *Also called cookie-based session*

- ***Server-side session***
  - *Data are stored in the server backend*
  - *Cookies store an **identification information***

- ***Common security issue***
  - ***Cross-site request forgery (CSRF)***
    - https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

Cross-site request forgery

- *Force execution of **unwanted actions***
  - *e.g., a javascript or a link with cross-site operations*

- ***Defense:***
  - https://www.owasp.org/index.php/CSRF_Prevention_Cheat_Sheet

- *"**CSRF token**"*
  - *Any operation that changes the state requires a secure random token*
    - *POST operations:*
      *Obtained via GET operations and added as **hidden field in forms***
    - *GET operations (discouraged):* ← *GET operations should NOT*
      Inserted via query parameters *modify the state*
- *Details in the security class*

CSRF in Flask [1]

- In some Web framework (e.g., Django) **CSRF Tokens are enabled by default**

- In Flask, you **must activate CSRF protection**
  - Can easily add them by using the **flask_wtf** library
  - http://flask-wtf.readthedocs.io/en/stable/csrf.html#csrf-protection

```
from flask_wtf.csrf import CSRFProtect
csrf = CSRFProtect(app)
```

  - Can generate the token in jinja templates by using the **`csrf_token`** method

- If non-GET handlers exist that **do not use session information**, you can exempt them from the token protection
  - **`csrf.exempt`** method, can be used as decorator

CSRF in Flask [2]

- The algorithm used to generate the CSRF token in Flask requires definition of a ***secret_key*** attribute in the WSGI object
  - ***Properly manage secret information required in your code***
    - ***Do not store them in the repositories***
    - ***Use different secrets in development and production environments***

- *GAE example:*
  - *Create a specialized Python module that maintains secret info*
  - *Can inspect GAE dev environment by using the SERVER_SOFTWARE environment variable*
  - *https://cloud.google.com/appengine/docs/standard/python/tools/using-local-server#detecting_application_runtime_environment*

  - `os.getenv('SERVER_SOFTWARE', '').startswith('Google App Engine/')`

Client-side sessions [1]

- *Client-side session example (multi-phase form compilation):*
  - *set-cookie: name=alice; surname=smith*

- **Performance trade-offs**
  - the server does not store anything
    - Any server can **recover a state →** useful in **load-balancing**
  - data are exchanged in each HTTP request → **bandwidth overhead**

- Important **security considerations**
  - what if clients **modify** data?
  - what if we (developers) store data the users should not access?

Client-side sessions [2]

- **Security issues**
  - what if clients **modify** data?
  - e.g., avoid previous input validation, fake authentication information
- **Defense:**
  - use cryptography to protect **data authenticity**
    - **Message Authentication Code** or **Digital Signatures**
      - *details in the security class*

- What **confidentiality of information** stored in the cookies?
  - Do we (developers) store critical data in the session?
    - Critical = that the user should not be able to read
      - e.g., server-side database information
    - **YES →** protect **confidentiality: encrypt** the data
    - **NO → authenticity is enough**

Concepts of MAC and Digital Signatures

- Message Authentication Coded (MAC) and Digital Signatures are cryptographic primitives to guarantee data **integrity** and **authenticity**
  - **Without knowledge of the secret key it is impossible to modify data without being detected**

- **MAC → symmetric setting**
  - Authenticate(key, data): MAC(key, data) → tag
  - Verify(key, data, tag): MAC(key, data) **==** tag (time const → digest_compare)
  - The keys used to authenticate and verify are the **same**

- **Signature → asymmetric setting**
  - Authenticate(sk, data): Sign(sk, data) → signature
  - Verify(pk, data, signature): Verify(pk, data, signature)
  - The keys used to authenticate (sign) and verify are **different**

Client-side sessions [3]

- ***Revocation* issues**
  - *Any self-contained crypto authenticated information should really store an **expiry** information*
    - *Details in the security course (x509, PGP, …)*
- ***Verification issues***
  - *Crypto implementations are difficult to implement and to use*
    - *e.g, metadata verification, allowed protocols, constant-time verification, …*
    - *https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/*
- ***Best practice: use existing libraries with good implementations***

Flask client-side sessions

- Flask **session** object
  - Standard dictionary interface to access data (key-value)
  - Also available by default in jinja templates
  - Use the *session.clear* to delete the session

- Flask uses client-side sessions by **default**
  - **Must** define the **secret_key** attribute of the flask wsgi object

Flask session exercise

- Use the *sessions_flask* application as a basis
  - Implement the *create_session* handler that adds the GET query arguments to the session object, and returns a page with all the contents of the session object
  - Implement the *clear_session* handler to clear the current session, if exists. Return a page that shows whether the session did not exist or if it has been cleared
  - Inspect how the session is implemented in the cookies

JWT and JOSE

- Standard to sign and encrypt serialized data in the Web
  - https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32
  - *"compact, URL-safe means of representing claims to be transferred between two parties"*

- *https://jwt.io/introduction/*
  - *JOSE header* **.** *{plaintext payload, JWE payload}* **.** *JWS signature*
  - *JWE → Json Web Encryption*
  - *JWS → Json Web Signature*
    - *Note: although called "signature", quite any service will use a MAC for performance reasons*

JWT in Flask

- Flask uses the ***itsdangerous*** *library to implement a* ***variant*** *of JWT*

- *Exercise:*
  - *Does Flask hide session information?*

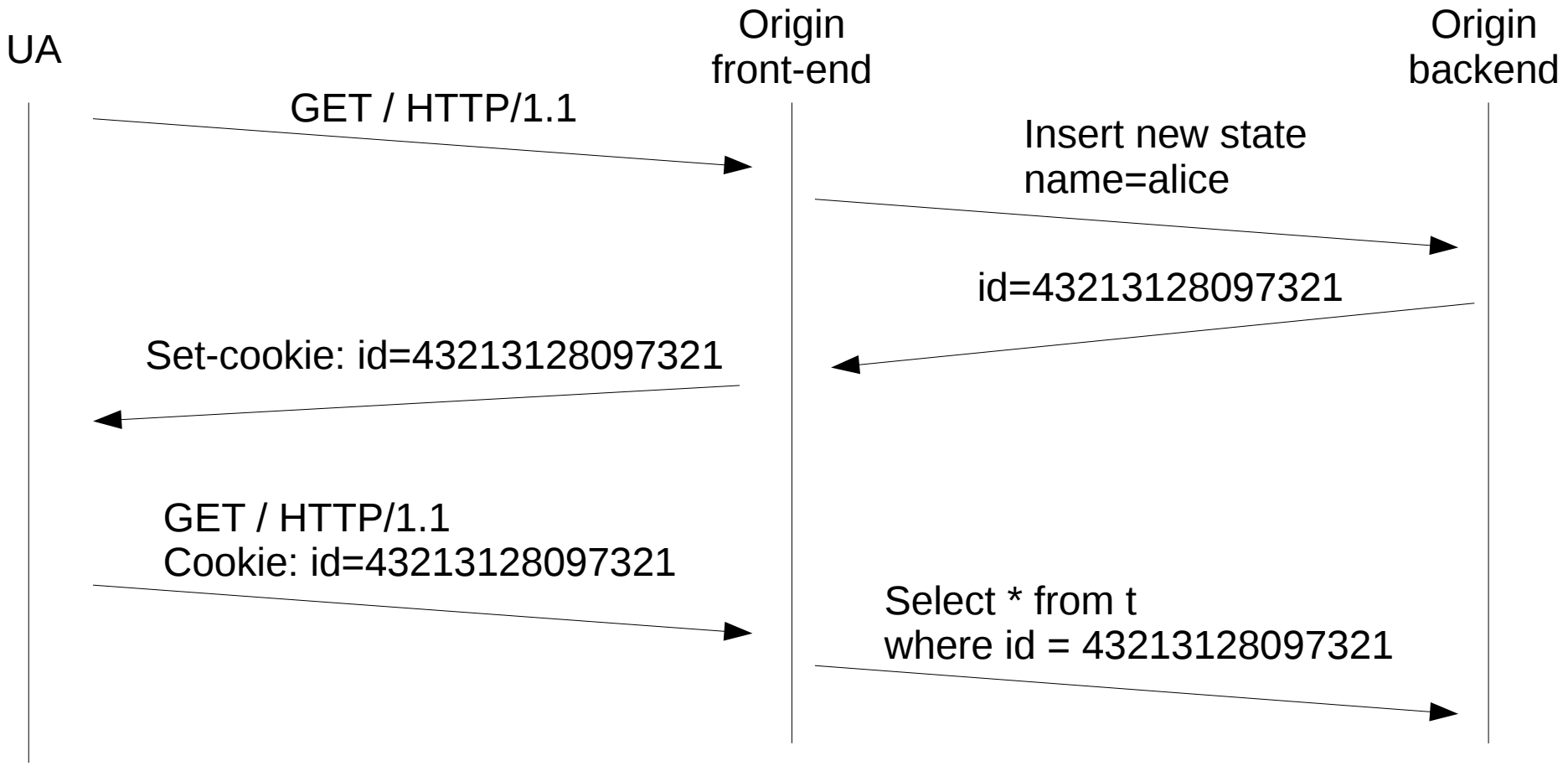
- *Extra* homework (send it by email):

  - produce Python code that, given the Flask session JWT and the server secret key, returns the information stored within

  - (*not trivial*) extend the session functionality with JWE to protect information confidentiality

Server-side sessions

- A typical approach to implement sessions is to use server storage and to exchange a reference to the state information



Server-side sessions on GAE

- Flask server-side sessions can be easily deployed by using the ***Flask-session*** library
  - https://pythonhosted.org/Flask-Session/
- However, Flask-session supports standard database interface
  - Can be used with memcached (sessions might unexpectedly disappear….)
- We suggest ***beaker***, a powerful stand-alone specialized library to handle **caching** and **state** information
  - https://beaker.readthedocs.io/en/latest/
  - Supports Flask sessions
  - Supports Google Datastore

Flask + GAE + Beaker server-side sessions

```
import beaker
from beaker.middleware import SessionMiddleware
from flask.sessions import SessionInterface

session_opts = {
    'session.type': 'ext:google',
    'session.auto': True,
    'session.httponly' : True,
    'session.secure' : True
}

class BeakerSessionInterface(SessionInterface):
    def open_session(self, app, request):
        session = request.environ['beaker.session']
        return session
    def save_session(self, app, session, response):
        session.save()

app.wsgi_app = SessionMiddleware(app.wsgi_app, session_opts)
app.session_interface = BeakerSessionInterface()
```

Homework

- Implement an application with authentication functions, with (minimal) registration and login forms
  - Authentication is handled via a **session**
  - Implement handlers that can be accessed only if the user is authenticated
- Do not forget
  - Cookies exercise (slide 5)
  - Session exercise (slide 16)
  - JWT extra-homeworks (slide 18)