CS-310 Scalable Software Architectures

Lecture 10: Authentication

Steve Tarzia

Last time: SQL Database Scaling

- **Read replicas** horizontally scale databases for reading.
  - Writes are done in one place and propagated to many replicas.
  - Data on a given replica may lag behind primary, but it's self-**consistent.**
  - Works well if writes are much less common than reads.
- Horizontal scaling of writes suggests **data partitioning**.
  - Each data row/element is assigned a single "home"
  - If not, consistency is very tricky (write race conditions for transactions).
- **Sharding** is data partitioning for SQL/relational DBs.
  - Works well for queries that can be handled within a single shard.
  - Sharding divides data along just one dimension, so inevitably some queries will involve all the nodes, and thus will not be scalable.

Authentication is proving your identity

- Most apps use require users to set a password to reconnect to account.
  - A *salted hash* of the password is stored in a database.
  - Passwords in future login requests can be compared to the stored password.
- An email address or phone number can also be used to prove identity.
- Web, desktop, smartphone, and other clients make requests to access user's data, and access must be protected.

- HTTP requests should be sent using HTTP**S** (TLS) which encrypts the data in transit. Request/response data cannot be intercepted.
  - TLS authenticates the *server* using certificates (details in CS-340 Networking).
  - However, TLS does not authenticate the *client*.

Simplest approach: Password in every request

Client can save the password locally and include it in every request.

- GET /inbox?user=steve&password=jordan23
- POST /message
  - Request body:

```
{ "user": "steve",
  "password": "jordan23",
  "to": "catlover3",
  "txt": "hello!"}
```

- Why is this a poor solution?
  - Storing the password locally is a security risk.
  - All the backend apps are seeing the password.
  - DevOps staff will see passwords in all HTTP request logs.

Session Keys are like temporary passwords

- When the user logs in, backend generates a random **session key**, stores it in the user account DB, and returns it to the client.
  - `POST /signin`
    Request body:
    ```
    { "user": "steve",
      "password": "jordan23" }
    ```
  - Response: 200 OK:
    ```
    { "session": "3lkjd0f9j32lkjsdef09j" }
    ```

We also call this an **Authentication Token.** Notice that we do not need the username. Why?

- Client includes session key in **all** future requests. Often in a **header**:
  - `GET /inbox`
    `Authorization: session 3lkjd0f9j32lkjsdef09j`

The second line is an optional http header.

Review: Cookies are auth tokens for web browsers

- Cookies are how web applications track **state,** often to track user identity.
- After user submits the login form, server will return a cookie in the response:

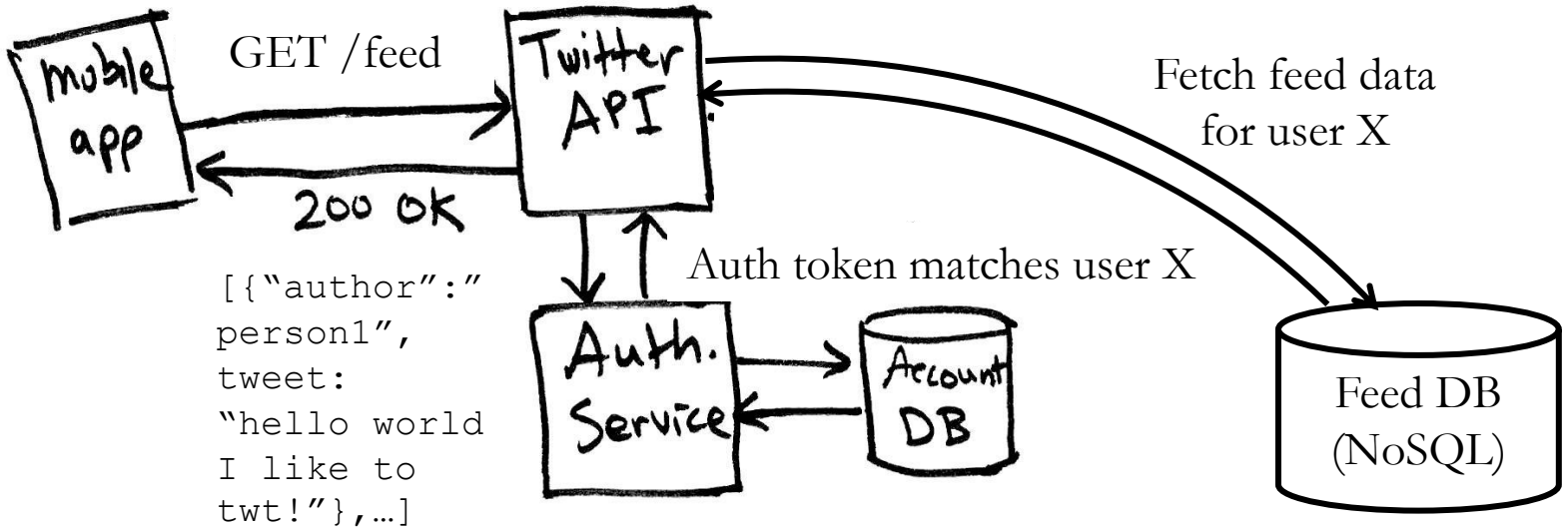
```
HTTP/1.1 302 Found
Location: http://somewebsite.com/account
Set-Cookie: someweb-id=kfj203d14t9s
```

- Response tells the browser to redirect to `http://somewebsite.com/account`, but it also gives the browser a cookie to remember.
- Browser will include the cookie in all future HTTP requests to `somewebsite.com`:

```
GET /account HTTP/1.1
Host: somewebsite.com
Referer: http://somewebsite.com/bin/login
Cookie: someweb-id=kfj203d14t9s
…
```

- Server getting this request can use the cookie to determine which user it came from!

Every request handler must now check the auth token



- The design above includes a separate microservice for authentication.
- Client device might send signin request directly to auth service.
- Other microservices ask the auth service to check auth tokens.

API Keys

- An authentication token that's valid for a long time is often called an **API key**.
- Eg. AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY
- 3rd parties may need programmatic access to your system.
- Your own backend microservices should authenticate with each other.
  - Use a local cache to check these quickly without reading from the DB.

Where to include auth token in a REST API?

Recall HTTP Request Inputs:

- Choice of Method:
  - GET/POST/PUT/DELETE
- Path
  - GET /tweets/**connor4real**
- Query parameters:
  - GET /search?startDate=2018-10-10&search=best+restaurant**&api_key=_____**
- Headers
  - Not recommended for normal parameters, but auth tokens are well-suited to headers.
  - Eg., **Cookie: _____** **Authorization: _____**
- Body
  - { **"session": _____,** … }

- Query param, Header, and Body are all reasonable choices.
- Headers are nice because they are separated from the request-specific parameters.

Two factor authentication

- For added security, some services require more than just a password.
- When handling `POST /signin` request, backend generates a random "challenge" code, stores it in a database, and sends to the user's known email or SMS address.
- User must click link or enter code to verify that they had access to the email account or phone to receive the secret.
  - "Forgot my password" feature also works like above.
- People tend to misuse passwords, so some services use email **exclusively** for login.
  - There is no password, and every login uses the "forgot password" style.

Recap

- Webservice requests are rarely open to the public.
- Each request must include an input that **authenticates** and identifies the user.
- Passwords are the most common auth mechanism.
- Email/SMS (a trusted *side channel* of communication) can be used.
- **Authentication tokens** are strings randomly generated (and stored) on the backend to verify user identity.
  - Variations include **session keys**, **cookies**, and **api keys.**
  - Often a separate microservice is dedicated to authentication (and other user management tasks, like account creation).
- **Digital signatures** are a more complex auth style, covered in CS-340.