Department of Computer Science
University of California, Los Angeles

Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

Lecture 10: April 30, 2025

Outline

Introduction

Today's topic is **State Management** which is a cryptic title that was short enough to fit on the syllabus.

HTTP is stateless. If we want to keep track of a user, user data, or a session, we have many ways to do it:

1. Server-Side / Persistent Storage
   - Relational Databases
   - MongoDB
   - Redis for Caching
2. Local State
   - Cookies
   - JSON Web Tokens (JWT)
   - Web Storage API

Persistent Storage

There are many situations where data needs to be stored long-term.

This is called persistent storage, and it includes file systems, cloud storage, and databases.

In this course, we'll focus on databases.

MongoDB

MongoDB Overview

- Document-based NoSQL database
- Stores JSON-like documents internally encoded as BSON
- Organized into databases → collections → documents
- Schemaless: documents in the same collection can differ in shape
- Popular in web development for flexibility and fast prototyping

Example of a Document

Below is an example of a document for a business from the Yelp Academic Dataset:


```
{
  "business_id": "gbQN7vr_caG_A1ugSmGhWg",
  "name": "Supercuts",
  "address": "4545 E Tropicana Rd Ste 8, Tropicana",
  "city": "Las Vegas",
  "state": "NV",
  "postal_code": "89121",
  "latitude": 36.099872,
  "longitude": -115.074574,
  "stars": 3.5,
  "review_count": 3,
  "is_open": 1,
  "attributes": {
    "RestaurantsPriceRange2": "3",
    "GoodForKids": "True",
    "BusinessAcceptsCreditCards": "True",
    "ByAppointmentOnly": "False",
    "BikeParking": "False"
  },
  "categories": "Hair Salons, Hair Stylists, Barbers, Men's Hair Salons, Cosmetics & Beauty Supply, Shopping, Beauty & Spas",
  "hours": {
    "Monday": "10:0-19:0",
    "Tuesday": "10:0-19:0",
    "Wednesday": "10:0-19:0",
    "Thursday": "10:0-19:0",
    "Friday": "10:0-19:0",
    "Saturday": "10:0-19:0",
    "Sunday": "10:0-18:0"
  }
}
```

MongoDB Data Model

- Each record is a **document** —- a JSON-like object with fields and values.
- Documents are stored in **collections**.
- Collections belong to a **database**.
- Documents are flexible —- no strict schema is required.
- MongoDB stores documents internally in a binary format called BSON.

MongoDB Data Model (contd.)

Feature	**JSON**	**BSON**
Format	Text-based	Binary (machine readable)
Data Types	Limited (no dates, no binary)	Rich (dates, ObjectId, binData, etc.)
Readability	Human-friendly	Compact and fast
Use in MongoDB	Accepted on input	Internal storage format

Note: Users interact with JSON — MongoDB handles conversion to BSON internally.

MongoDB: Queries

MongoDB supports CRUD:

- **C**reate: `insertOne()`, `insertMany()`
- **R**etrieve: `findOne()`, `find()`
- **U**pdate: `updateOne()`, `updateMany()`
- **D**elete: `deleteOne()`, `deleteMany()`

MongoDB: Queries (contd.)

To find data, we use the `find` or `findOne` query using the following template:

```
db.Books.find(
    // Selection condition (WHERE). Find programming books.
    { "category": "programming" },
    // Projection operator. Hide _id, show title (0/1)
    { "_id": 0,  "title": 1 }
)
    // sort by title ascending (vs -1 descending), and limit.
.sort({ "title": 1 })
.limit(5);
```

MongoDB: Queries (contd.)

Aggregation Pipeline: *(Advanced — not required for this class.)*

```
db.Books.aggregate( [
  // Match stage (WHERE).
  {
      "$match": { "category": "programming" }
  },
  // Group stage (GROUP BY). Can use multiple functions.
  // use _id: null for global aggregate.
  {
      "$group": { "_id": "$language", "AvgPrice": { "$avg": "$price" } }
  },
  //optional Projection
  { "$project": { "_id": 1, "AvgPrice": 1} },
  // optional Sort stage
  { "$sort" : { "AvgPrice": -1 } },
  // optional Limit stage
  { "$limit" : 10 }
] );
```

MongoDB: Queries (contd.)

Check out the README in the class Github repo if you want to follow along (under lecture 10).

Let's do some examples using the Yelp Academic Dataset to answer some questions:

1. Retrieve all Starbucks, their states and ratings.
2. Retrieve all restaurants that meet my casual but picky needs.

Scaling MongoDB for Large Applications (Reference Only)

Problem: As your application grows, a single MongoDB server may run out of CPU, RAM, or disk.

Solution: Horizontal Scaling with Sharding

- MongoDB can split collections across multiple machines (called **shards**).
- A special query router (`mongos`) handles routing requests to the right shard.
- This lets apps scale beyond the limits of one machine.

Note: You won't need this for CS 144, but it's good to know what's possible in production-scale systems.

MongoDB Replica Sets (Reference)

MongoDB supports high availability through **replica sets**:

- One **primary** node handles all writes and (by default) reads.
- One or more **secondary** nodes replicate data asynchronously.
- If the primary goes down, a new primary is elected (*automatic failover*).
- Optional **arbiters** vote in elections but store no data.

Why this matters: Your app can survive failures — but some reads or writes may be delayed briefly.

MongoDB and the CAP Theorem (Reference)

Question: Given what you know now, where does MongoDB fall under CAP by default?

- Writes (and reads) go to the primary this **strong consistency**
- Secondaries are async thus **no guarantee of up-to-date reads**
- If the primary is down, there's a pause for election **partition tolerance**

MongoDB and the CAP Theorem (Reference) (contd.)

But: MongoDB can also be configured to allow reads from **secondaries**.

Question: Now what?

Notable Features of MongoDB

MongoDB supports several features beyond basic CRUD:

- Geospatial indexes — for mapping and location-based queries.
- Full-text search — powerful for matching keywords in documents.
- Flexible aggregation pipelines — for data transformation and reporting.
- Capped collections — fixed-size collections useful for logs or telemetry.
- TTL indexes — documents auto-expire after a set time.

These last two make MongoDB a reasonable choice for some document-based caching use cases.

MongoDB: Common Criticisms (Past and Present)

While MongoDB is widely used today, it received significant criticism in its earlier years:

- Multiple, inconsistent access layers (Query, MapReduce, Aggregation).
- Piecemeal design — some engineers called it a "Frankenstein database."
- Historically poor default security (e.g., unauthenticated installs).

Many criticisms have since been addressed — but data modeling still requires care.

Document Modeling Limitations

"Whether you're duplicating critical data (ugh), or using references and doing joins in your application code (double ugh), when you have links between documents, you've outgrown MongoDB."
– *Sarah Mei*

This reflects common challenges with deeply nested documents or relational-style needs. Joins are possible via `$lookup`, but can be clunky for complex schemas.

Interacting with Databases

Modern web frameworks often provide object-oriented interfaces to databases:

- **ORM (Object-Relational Mapping)** -– Maps objects to rows in a relational database (e.g., Sequelize, Prisma)
- **ODM (Object-Document Mapping)** — Maps objects to documents in a document database (e.g., Mongoose for MongoDB)
- **OGM (Object-Graph Mapping)** — Maps objects to nodes and edges in a graph database (e.g., Neo4j's OGM)

These tools abstract database operations and enforce structure while letting you work in your application's native language.

Relational Model: ORMs by Ecosystem

Object-Relational Mappers (ORMs) exist across most major programming ecosystems:

Python

- **SQLAlchemy** — Lightweight, flexible ORM
- **Django ORM** — Comes bundled with Django web framework

PHP

- **Laravel** — Includes Eloquent ORM
- **CakePHP, Yii, CodeIgniter** – All ship with built-in ORM layers

Java

- **Hibernate** -– The most popular ORM for Java

C# / .NET

- **Entity Framework**, **Dapper** -– Two major ORM options

ORMs let you interact with a relational database using language-native objects instead of raw SQL.

ODM: Mongoose

The most popular ODM for MongoDB is Mongoose.


```
npm install mongoose
```

ODM: Mongoose (contd.)

Mongoose lets you:

- Define schemas and models for MongoDB collections
- Perform validation and type enforcement
- Simplify querying and CRUD operations

Mongoose operations are **asynchronous**, so we'll use `async/await` when interacting with it.

Connecting to MongoDB

```
import mongoose from 'mongoose';

// Optional: disable strict mode for query filters
mongoose.set("strictQuery", false);

const mongoDB = "mongodb://ip_address/mammoth";

async function main() {
    await mongoose.connect(mongoDB);
}

main().catch((err) => console.log(err));
```

Once connected, the connection is available as `mongoose.connection`.

Create a Model

MongoDB is schemaless. We are going to define our data model – how we are going to fill data structures with data from MongoDB.

To do that, we need a schema within the model.

Each schema/model should be in its own file and imported.

```
import mongoose from 'mongoose';

// DISCLAIMER: This will not exactly match project 2.
const LiftSchema = new mongoose.Schema({
    name: { type: String, required: true },
    status: {
        type: String,
        enum: ['OPEN', 'CLOSED'],
        default: 'CLOSED',
        required: true
    },
    capacity: {
        type: mongoose.Schema.Types.Mixed,
        validate: {
          validator: (v) => typeof v === 'number' || v === 'Gondola',
          message: 'Capacity must be a number or the string "Gondola".'
        }
    },
    misc: String,
    lastUpdated: { type: Date, default: Date.now }
}, { _id: false });  // These are only used a subdocuments, no need for OID.

const Lift = mongoose.model('Lift', LiftSchema); // call find etc. on Lift.
export default Lift;
```

Create a Model

Mongoose schemas support a rich set of features for shaping and validating data:

1. Multiple built-in data types (see SchemaTypes)
2. Default values for fields
3. Built-in validators (e.g., `min`, `max`) and custom validation functions
4. Required fields
5. String modifiers (`lowercase`, `trim`, `match`, etc.)
6. Virtual properties (computed values not stored in the database)

Create a Model (contd.)

Now we can insert data into the model:

```
const liftInstance = new Lift({
  name: "Broadway Express",
  status: "CLOSED",
  capacity: 6,
  misc: "high-speed",
  lastUpdated: new Date()
});

// Save the new model instance asynchronously
await liftInstance.save();
```

The call to `save()` persists the document to MongoDB.

Create a Model (contd.)

Fetching looks familiar:

Object-style query

```
const lift = await Lift.find({
  status: "OPEN",
  capacity: { $gt: 2 }
}).exec();
```

Chained query builder

```
const lift = await Lift.find()
  .where("status").equals("OPEN")
  .where("capacity").gt(2)
  .exec();
```

Also works with `findOne()`

Both return the same result —- choose the style that's more readable for your use case.

Create a Model (contd.)

You can update a document by fetching and saving, or using a direct query.

Fetch → Modify → Save
(validates + triggers middleware)

```
const lift = await Lift.findOne({
  name: "Broadway Express"
}).exec();
lift.status = "OPEN";
await lift.save();
```

Direct Update
(no validation by default)

```
await Lift.updateOne(
  { name: "Broadway Express" },
  { status: "OPEN" }
);
```

Use the first method if you rely on schema validation or lifecycle hooks.

Create a Model (contd.)

Web applications typically use ORMs, ODMs etc.

It's also possible to use a native client for writing raw queries, like we did in 143.

Key-Value Stores

Key-Value Stores come in all sorts of flavors. The three most common key-value stores are as follows. I also add two more interesting ones: Dynamo and LevelDB/RocksDB.

1. MemcacheDB (memcached with persistence) / memcached
2. Berkeley DB (MemcacheDB relies on Berkeley DB)
3. Redis
4. Riak
5. Dynamo (storage system that powers DynamoDB)
6. LevelDB and RocksDB

Redis

Redis stores values by keys, but can store *data structures*.

Redis is consistently rated the most popular key-value database, and the #4 NoSQL database.

Redis actually stands for "REmote DIctionary Server." The more you know! It has bindings with every modern programming language, which has helped it remain successful.

Redis (contd.)

Redis offers the following data types that many of other systems do not support natively:

1. Single key-value pairs, lists, sets and sorted sets (of strings)
2. Hash tables where keys and values are strings (classic key-value pair)
3. HyperLogLog and Bloom Filter!
4. Streams
5. Geospatial (geohash) including radius queries

Compared to other key-value databases, each data type has different methods we can apply.

Redis (contd.)

Typical Redis uses include:

1. session caching (big one)
2. message queues (I use it for this mostly)
3. leaderboards
4. fast lookup and indexing

Redis (contd.)

Commands are executed via an API, or directly via Telnet or via the Redis CLI in your favorite language.

There are too many commands to mention, but they all take either a key and a value, or just a key. Let's do a demo.

Redis (contd.)

For your project 2, you will use Redis as a cache to speed up retrieval of ski resort information. The cache will be restricted in size as that is usually how caches work.

For items that are not in the cache, your API will need to fetch the data from MongoDB. You can see that there may be some inconsistencies here.

All writes and mutations will be in your cache rather than MongoDB.

Redis (contd.)

Redis has many other uses than just as a cache:

1. Pub/Sub architecture for real-time notifications
2. Rate limiting
3. Streaming data
4. Even vector search
5. Counters

Redis (contd.)

Some cloud-based key-value stores similar to Redis:

1. Amazon ElastiCache (this actually is Redis hosted on AWS)
2. Google Cloud Memorystore (again, this is Redis hosted on GCP)
3. Upstash (serverless Redis with HTTP API)

References

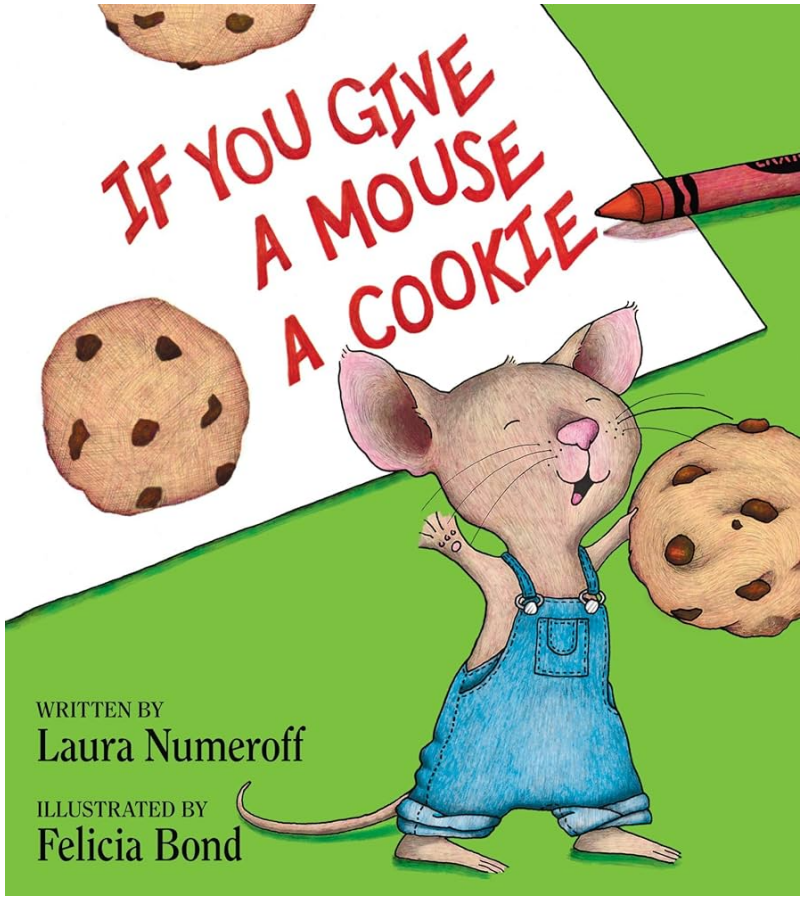
- MongoDB Documentation
- Mongoose Documentation
- Mozilla Express + Mongoose Tutorial
- Redis Documentation

Cookies


IF YOU GIVE
A USER AGENT
A COOKIE
GET / HTTP/1.1
Host: example.com
Set-Cookie:

Cookies (contd.)

Inspired by...



Cookies (contd.)

HTTP is **stateless**. Once a response is sent from the server to the client, there is no memory of the request/response pair anymore. So...

- How do websites know who we are? (e.g. identity)
- How do they know our preferences? (e.g. favorite book genres)
- How do they remember site settings? (e.g. locale, light/dark mode)

One way is by creating a "primary key" and somehow embedding it into every HTTP request.

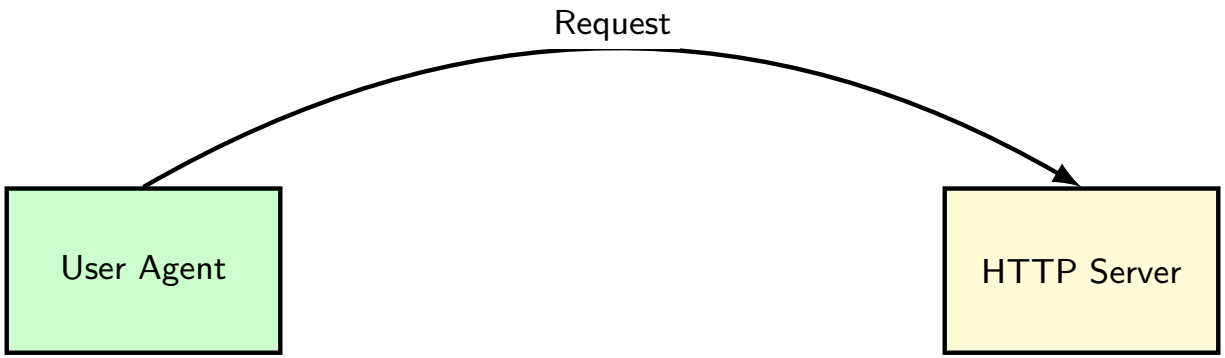
Cookie Exchange

Cookies allow the server to ask the client to remember a series of `name=value` pairs.

Once the cookie is set, the client always sends the cookie back to the server on every subsequent HTTP request, even if it is not needed.

Cookie Exchange

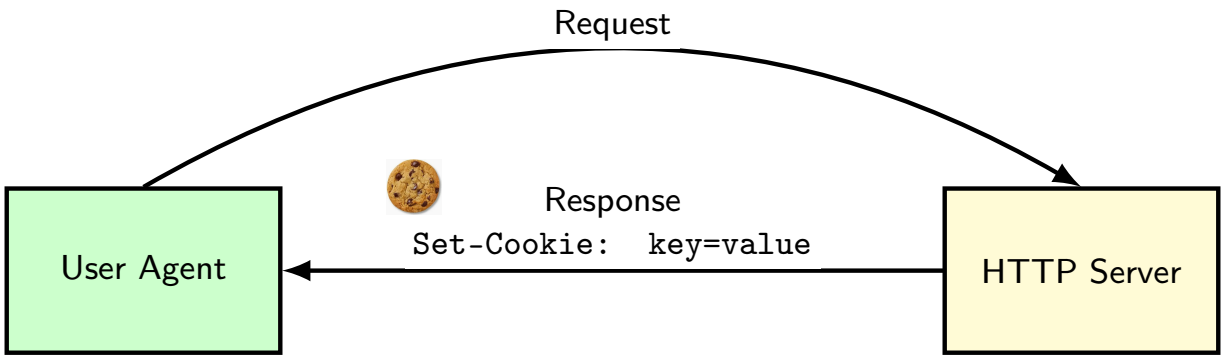
Cookies allow the server to ask the client to remember a series of `name=value` pairs.



Once the cookie is set, the client always sends the cookie back to the server on every subsequent HTTP request, even if it is not needed.

Cookie Exchange

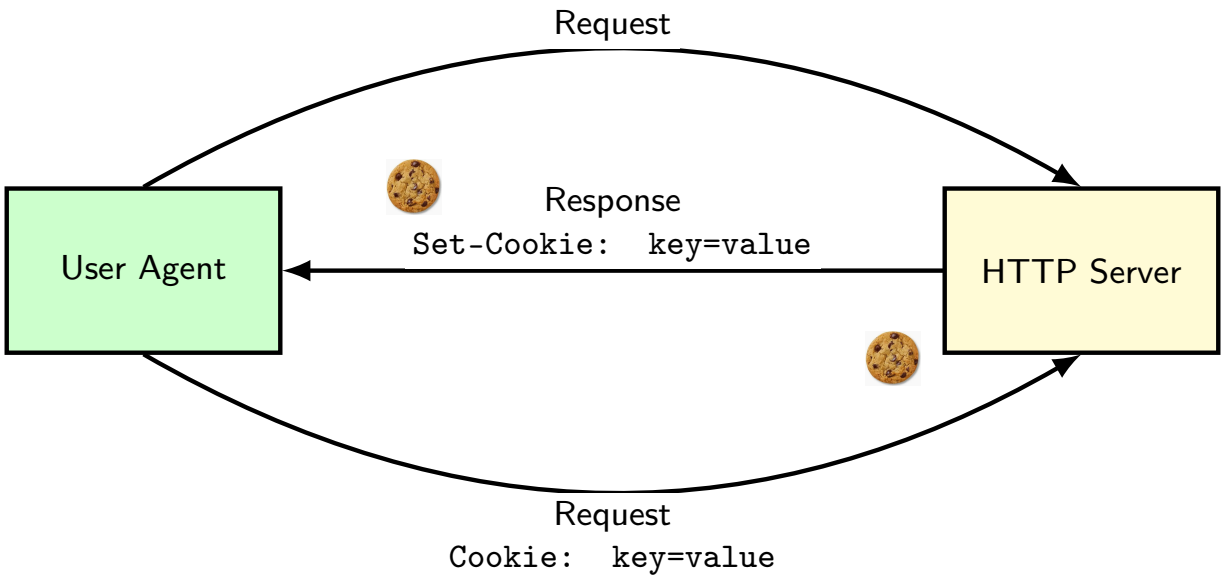
Cookies allow the server to ask the client to remember a series of `name=value` pairs.



Once the cookie is set, the client always sends the cookie back to the server on every subsequent HTTP request, even if it is not needed.

Cookie Exchange

Cookies allow the server to ask the client to remember a series of `name=value` pairs.



Once the cookie is set, the client always sends the cookie back to the server on every subsequent HTTP request, even if it is not needed.

How to Set a Cookie?

The server asks the client to set a cookie using the HTTP response:

```
Set-Cookie: sessionabc123; Path=/; HttpOnly; Secure; SameSite=Lax; Max-Age=3600
```

`session=abc123` is the name/value pair to save and `expires` indicates when the cookie will expire. If there is no `expires` field, the cookie is only valid for the current session.

Usually cookies are valid for an entire server, but they can be restricted to certain paths.

How to Set a Cookie? (contd.)

In all future HTTP requests to the specified domain and path, the cookie is sent in a header.

```
Cookie: session=abc123
```

Client-side can, in some cases, set or modify cookies using JavaScript:

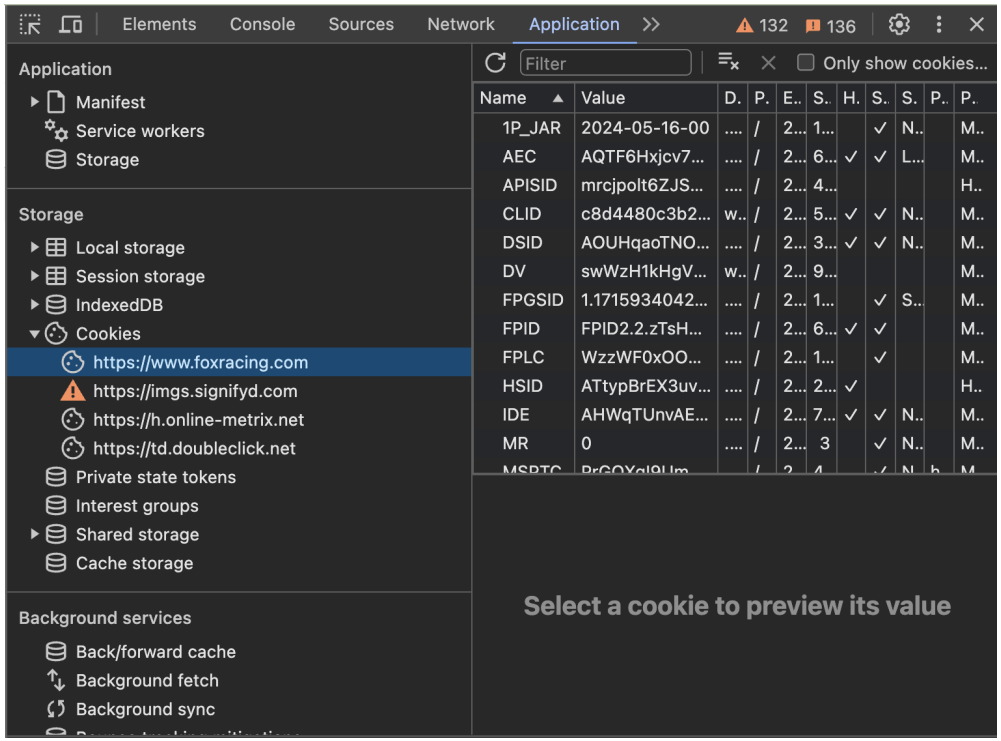
Same-Origin Policy

The user agent only sends a cookie in an HTTP request to the domain that originally set the cookie.

Why? If cookies from site *A* could be sent to any other domain on the Web, we have a big privacy and security concern. A cookie should be a "secret" (kinda/sorta) between the client and server.

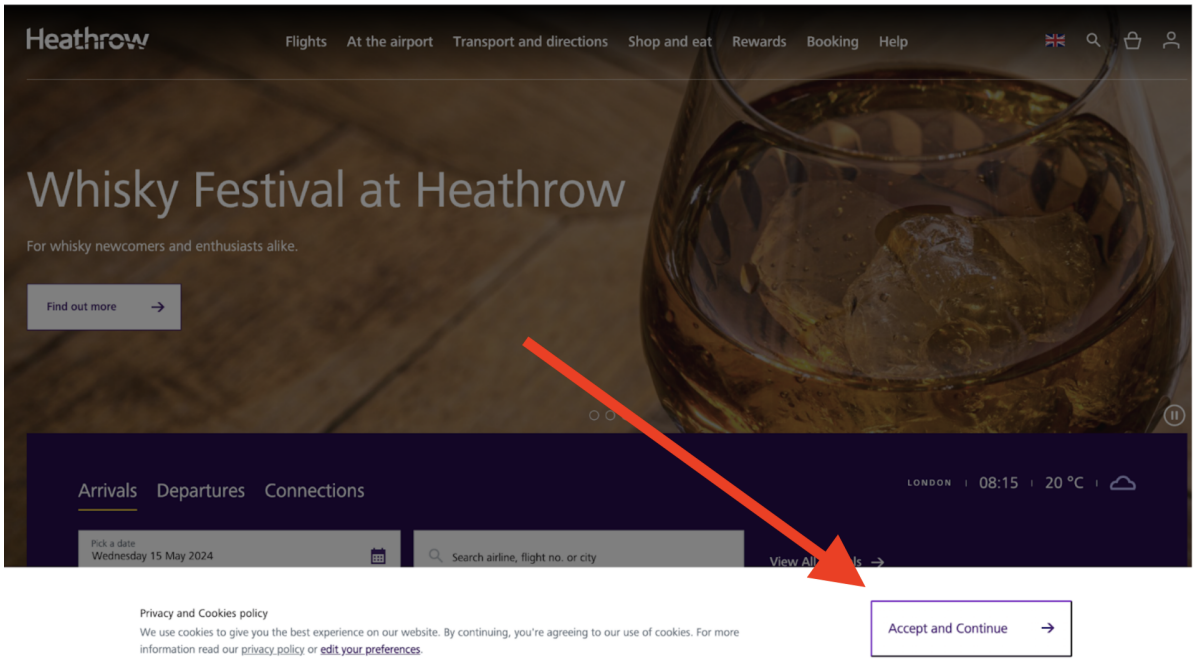
Hands in the Cookie Jar

You can see what cookies are set for a particular domain using Chrome Developer Tools: Applications > Cookies:



Do *You* Want a Cookie?

Two privacy regulations, GDPR (EU) and CCPA (California), requires users to consent to accepting cookies. That's why you see the following banner everywhere:



Cookies are Risky

Cookies can be stolen (cookie theft) from an innocent user A and the thief, roque user B, can masquerade as A.

Possible improvements:

1. Add `HttpOnly` to `Set-Cookie`. JS can't access (XSS).
2. Add `Secure` to `Set-Cookie`. Only sent over HTTPS. (MITM).
3. Add `SameSite=Lax` to `Set-Cookie`. (CSRF).)

Cookies are Risky (contd.)

The user agent, or a malicious script executed by it, can also tamper with the contents of the cookie (cookie poisoning). For example changing `Set-Cookie: role=user` to `Cookie: role=admin`.

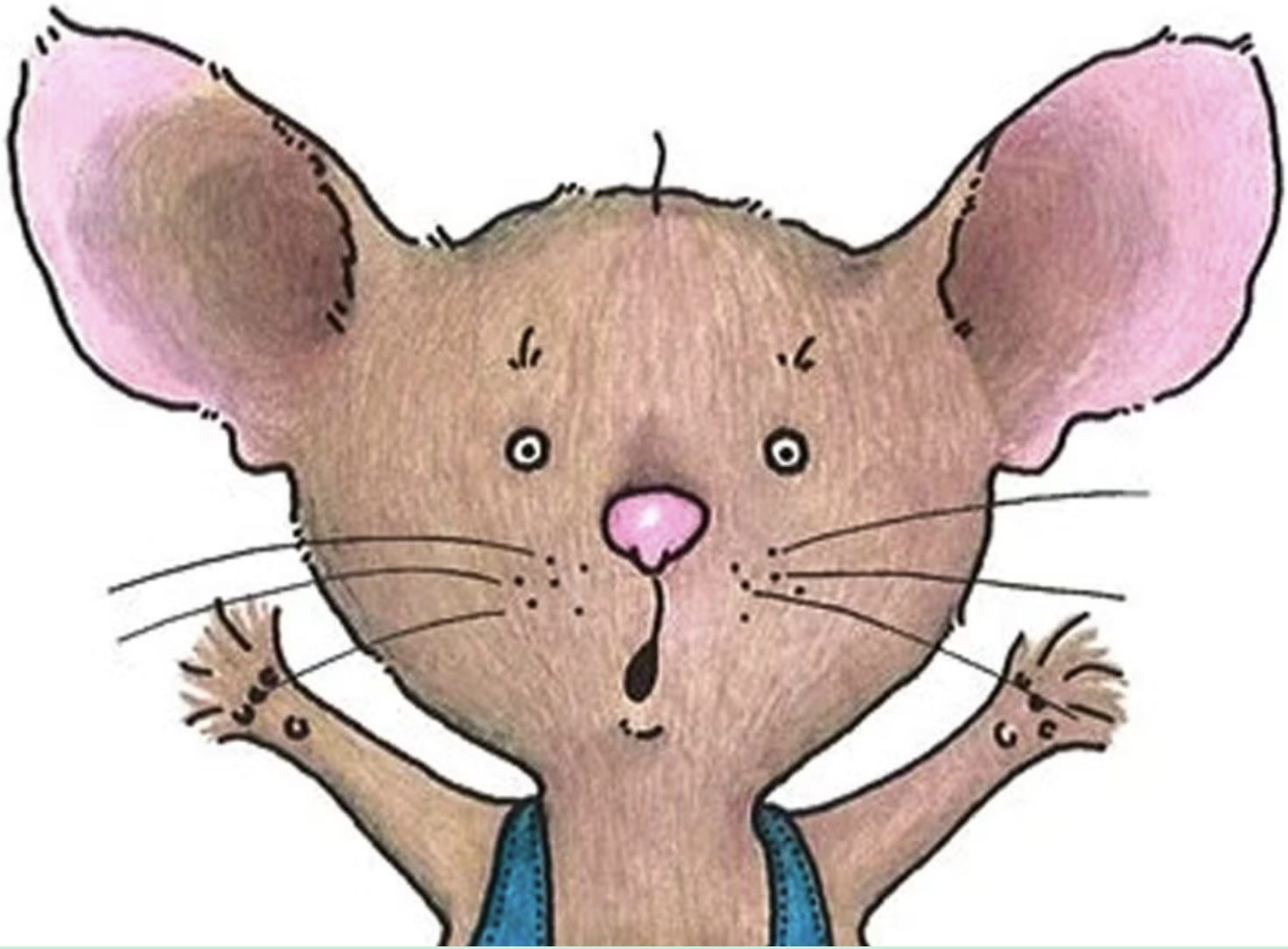
Possible improvements:

1. Server signs the cookie:
2. Client sends the cookie with each request.
3. If signature doen't match, server rejects cookie.

Cookies are Risky (contd.)

That is, there are too many dirty hands (or paws) in the cookie jar.

Cookies are Risky (contd.)



Keeping Your Cookies Safe

As a developer, you need to be careful about what you store in a cookie and how you handle it.

Keeping Your Cookies Safe (contd.)

There are a variety of other ways to protect cookies:

1. Set short expirations or `Max-Age` values. `Set-Cookie: sessionId=abc123; Max-Age=1800`
2. Avoid path and domain overexposure: `Set-Cookie: sessionId=abc123; Path=/api`.
3. Regenerate session IDs on login and logout.
4. If the cookie is set by local JavaScript, sanitize all inputs to prevent XSS (use DOMPurify)
5. Store sensitive data server-side and only server-side

On point #1, even if the cookie is stolen, a stale, expired cookie is no fun to eat...

Keeping Your Cookies Safe (contd.)

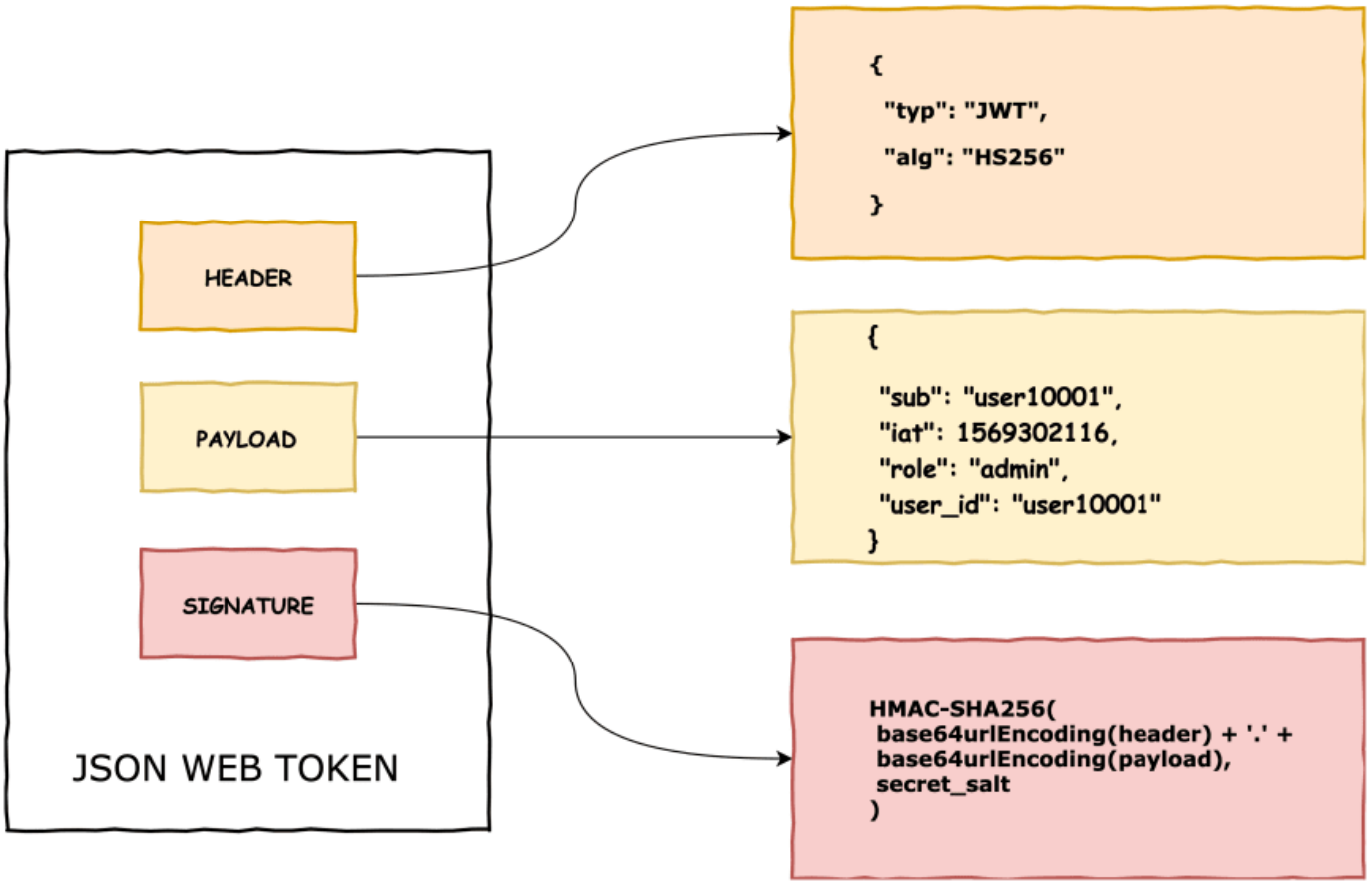
JSON Web Tokens (JWT)

JSON Web Token (JWT) is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. It looks like:

```
header.payload.signature
```

Header: describes the token (usually type `JWT` and algo like `HS256`).
Payload: contains claims like `sub` (user ID), `exp` (expiry), etc.
Signature: HMAC or RSA signature verifying the token hasn't been altered.

JSON Web Tokens (JWT) (contd.)



Source

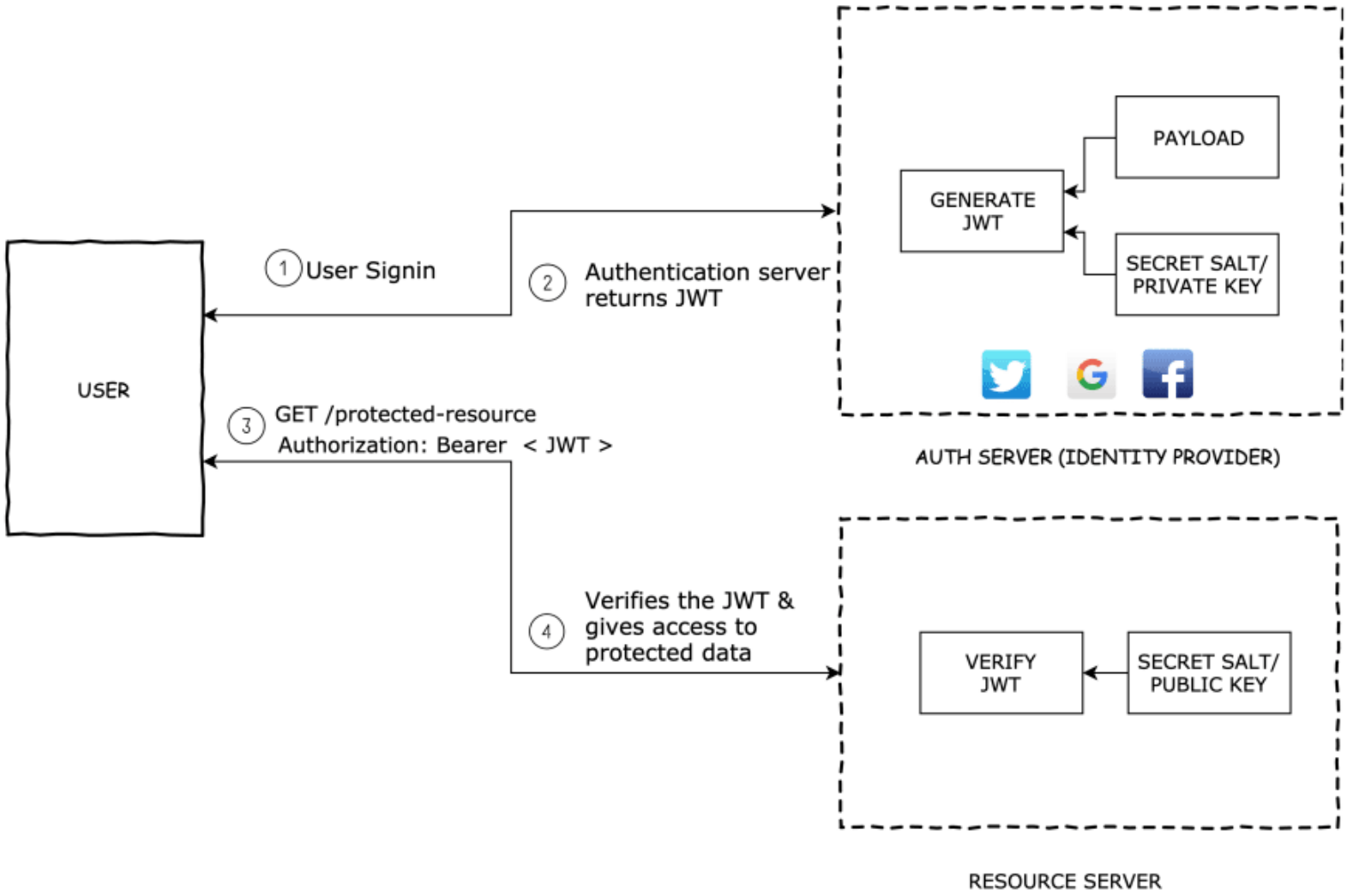
JSON Web Tokens (JWT) (contd.)

Example Flow:

1. User logs in and the authentication server returns a signed JWT, typically stored in a cookie or client-side storage.
2. On each request, the JWT is sent via the `Authorization: Bearer <token>` header or via an `HttpOnly` cookie.
3. The server verifies the token's signature and claims using the shared secret or public key.

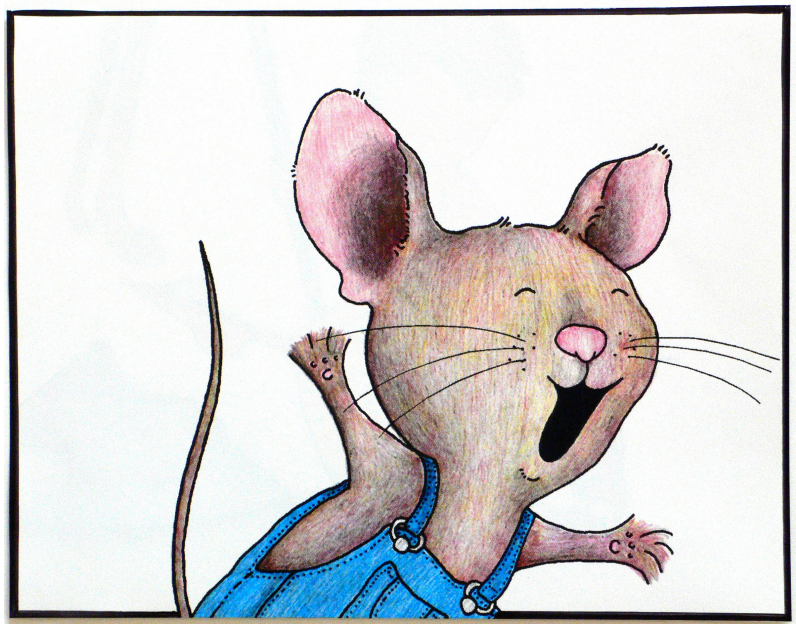
Key Point: JWTs are *not encrypted* — they are *signed*. Anyone can read them, but only the server can validate them.

JSON Web Tokens (JWT) (contd.)



The End of Cookies

And the mouse lived happily ever after with no cats stealing cookies or humans poisoning them.



User Authentication

How does a server authenticate a user and recognize them on future requests?

1. User logs in with credentials.
2. Server verifies credentials and creates a session (e.g., in Redis).
3. Server sends back a cookie with a unique `session_id`.
4. On future requests, the browser sends the cookie.
5. Server looks up session data using the `session_id`.

This is called **stateful authentication**.

User Authentication (contd.)

We typically avoid storing user identity directly in the cookie.
Instead, store:

- A unique, unguessable `session_id`
- User data lives server-side (e.g., Redis or database)

Why not store the username directly in the cookie?

- Easy to spoof
- May expose personally identifiable info (PII)
- Safer to use opaque, signed session tokens

Session vs JWT-Based Authentication

Feature	**Session ID (Stateful)**	**JWT (Stateless)**
Storage	Server-side (e.g., Redis or memory)	Client-side (in cookie or Authorization header)
Cookie Content	Opaque `session_id`	Readable JSON claims like `sub`, `exp`, `iat`
Scalability	Requires shared session store across servers	Stateless — scales easily without server storage
Revocation	Easy — delete session from backend store	Hard — must expire or be blacklisted
Security Risk if Stolen	Session can be hijacked, but identity is not exposed	Claims are exposed; must verify signature
Use Cases	Traditional web apps, server-rendered flows	APIs, SPAs, mobile clients, federated auth

Note: JWTs in cookies should use `HttpOnly`, `Secure`, and `SameSite=Lax` for best

WebStorage API

The Web Storage API provides mechanisms by which browsers can store key/value pairs in a much more intuitive fashion than using cookies.

Until HTML5, most storage was server-side in a database, with other data being exchanged between the client and the server using a cookie.

WebStorage API (contd.)

Browsers now support local storage that client-side scripts can read and send to a server; servers cannot access it directly.

1. `sessionStorage`: Per-origin storage that lasts only while a browser tab is open. Data is cleared when the tab or window is closed. Ephemeral.
2. `localStorage`: Per-origin storage that persists across page reloads, tab closures, and browser restarts—until explicitly cleared by the user or JavaScript.

Both have a standard key-value interface.

WebStorage API (contd.)

The API is simple to use:

```
window.localStorage.setItem('appMode', 'light');

window.sessionStorage.setItem('appMode', 'light');
let user = window.sessionStorage.getItem("user");
window.sessionStorage.removeItem("user");
```

Each origin has its own storage and storage sizes are small, at about 10MB and there are still similar reliability issues as with cookies.

WebStorage API (contd.)

Never store authentication tokens (e.g., JWTs) in `localStorage` or `sessionStorage` unless you fully control your frontend and trust its scripts.

These APIs are accessible via JavaScript, which means:

- They are vulnerable to **XSS** (Cross-Site Scripting).
- An attacker injecting JS can steal sensitive data from storage.

Instead store them in a cookie with `HttpOnly` and `SameSite=Lax` flags.

The Future

Serverless functions and edge computing are changing how state is managed.

In serverless environments, traditional in-memory session storage (like express-session defaults) won't work —- because you don't control the server process.

Instead, sessions are typically stored in an external system or encoded in JWTs. Yes, specifically Redis, because of the number of features it has for a cache and because it is compatible with so much other services like Vercel and Cloudflare.

References

- HTTP Cookies
- Web Storage API
- JSON Web Tokens