In a web 1.0 world this was very difficult. Few sites had an API, so implementing this functionality was very difficult. Often you would need to effectively “log in” as the user (which required the user to share their password). This precluded collaboration between services. Increasingly services like Facebook and others provide APIs that facilitate this integration. In this model, which I suspect you have used extensively, when an application wants to access data held by a 3rd party, e.g. Facebook, it redirects the user to that 3rd party to login directly (authenticate) and authorize the application (e.g. NYTimes) to access the user’s data. If the user agrees, Facebook, or other 3rd party, sends back a token that the application (NYTimes) can use to retrieve just the data its authorized to access.

Note the distinction between authentication and authorization, the user authenticates with Facebook, but authorizes the NYTimes to access certain resources.

Single sign-on is a variant of this approach (without the data sharing), where we rely on authentication provided by a third party, e.g. Microsoft or Google, to verify the person is the same person who initially setup the account... Unless there is a compelling reason for an alternate approach, any projects that need authentication should use a 3rd party (specifically Google). This is an aspect where we want to avoid DIY if we can... it is just to easy to get it wrong.
Reminder: Never trust the client

The user has total control over their browser
    Can bypass any “protections” you built into app
Or could access your API end points directly
    Your JS isn’t needed to make HTTP requests
Thus any validation, authentication and authorization must be performed on a/the server (or with its assistance)

For example, in Simplepedia we disabled the Save button, but that is just an HTML attribute. The user could easily “re-enable” that button in their browser.

When we have tested our servers, we have used the curl command line tool, that is no browser involved. Someone with malicious intent could similarly make requests to your server without involving your front-end application in any way. So you can’t rely on any “protections” built into that application.
We want to distinguish between authentication, proving your identity, and authorization, proving that you are allowed to perform some action. This is a familiar notion to us, that different users may have different roles, or privileges, and that just because I have an account on a service, just because I am authenticated doesn’t mean I can access all data, routes, etc.

In many cases authentication is a prerequisite to authorization, that is first I login, then I can access certain data/features. However, in widely used 3rd party workflows, i.e. where an application is accessing data held by a third party, that application uses a cryptographic token, issued by the 3rd party, to prove the user has authorized it to access that data, etc.

In our practical we will implement both aspects, i.e. authn and authz, albeit with simple authorization – some actions are restricted to logged in users (but we don’t distinguish between different users). In many of your applications you will need more sophisticated approaches to protect sensitive data.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
In the example we use a cryptographic signature algorithm, with a secret key known only to the server, to “sign” the data payload, that is effectively compute a hash. Any change to the data payload will change the signature, and thus we can detect any change the signature. And since the key is a secret, any attempt to generate a fake token, i.e. generate a fake signature will fail validation with the real key. There many variations on this general approach using different cryptographic primitives, including public-private key encryption, by they share the common feature that the server can detect if the client has manipulated the token.

What is in the data payload? Typically just a “handle” or pointer to data the server maintains, e.g. unique identifier for the user, session, etc. That is the data in the payload is not typically itself sensitive.

What is a fundamental limitation of this approach? The token is all the requestor needs to obtain the protected resource. Thus the client has to protect that token.
Because we are implementing a single page application (SPA), we will use a workflow where the browser communicates directly with Google to initiate the login. After the user authenticates with Google, Google sends back a token that can be used by the server to validate the user. The token is tamper evident (similar to the mechanisms we just described) and so can be validated/verified with Google. Once the server has validated the token, it will create an account for the user if they don’t have one already and create a session for the user to authenticate future interactions.
To implement this workflow we will use the Passport library, which provides a variety of workflows. Passport provides a set of middleware that we can integrate into our Express request pipeline. Behind the scenes we are effectively create a new model “Session” that stores the necessary data to identifying the user, typically the user id. Effectively the session “remembers” the user. These sessions are typically stored in the database (although in our practical we will store them in memory to keep it simple).
### Managing statelessness: Cookies

- **Observation:** *HTTP is stateless*
- Early Web (pre 1994) didn’t have a good way to guide a user “through” a flow of pages...
  - IP addresses are shared
  - Query parameters hard to cache, makes URLs private information
- **Quickly superseded by cookies**
  - Set by server, sent by browser on every request
  - Since client-side, must be tamper evident

  **Remember: Never trust the client!**

What is the value of statelessness? Treats requests independently. No need to maintain client’s previous interactions, and thus different servers can handle different requests.

We use similar cryptographic approaches to what we described previously for ensuring that cookies are tamper evident.

Note that we switch to using cookies because we are authenticating with our own server. Cookies are only sent to the server that set them, so when interacting with 3rd party APIs we will continue to use tokens. We could also use tokens with our own API (may be needed if not clients are not a browser), but I think we will find cookies easier, since they are transparently handled by the browser and automatically sent by fetch (in more recent versions of the specification).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Which of the following is true about 3rd party authn and authz between requestor and a provider?

A. Once completed, the requestor can do anything you can do on the provider
B. If your login credentials on the requester are compromised, your login credentials on the provider are also compromised
C. If the provider revokes access, the requester no longer has any of your info
D. Access can be time-limited to expire on/after pre-set time

Answer: D

We can approach this via elimination, that is A-C are all false. The requestor is allowed to do those operations you have authorized, not everything you can do. The requestor doesn’t have your login credentials (just a token) and so a breach at the requestor doesn’t compromise your login credentials at the provider. And while you can revoke future access, you can’t “revoke” access to data the requestor has already obtained. We can however can typically set access to only last for a certain amount of time before it expires and needs to be renewed (or is terminated).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Recall that with public-private key encryption, the public key can decrypt messages communicated with private and vice versa. I give out the public key widely so that partners can decrypt my messages (and know those messages are from me) and encrypt messages to me (which only I can read). The browser and server use key exchange methods to bootstrap this encrypted channel after verifying that the server’s certificate was signed by a trusted certificate authority. That indicates that server is who it says it is...

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
What SSL does and does not do

✓ Prevents eavesdropping on traffic between browser and server
✓ Assure browser that the server is legitimate (for some value of legitimate)
✗ Validate identity of user
✗ Protect data after it reaches the server
✗ Ensure server doesn’t have other vulnerabilities
✗ Protect browser from malicious server
Securing our applications

There are many potential vulnerabilities

- Eavesdropping
- (SQL) injection
- Man-in-the-Middle/Session hijacking
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)

And much more...

Some of these we have talked about, e.g. eavesdropping and how we could use SSL to mitigate that risk. But many more we won’t discuss or only touch on briefly. For example...
Example: SQL injection

```sql
knex.raw(`SELECT * FROM Article WHERE id = ${id}`);
```

```sql
SELECT * FROM Article WHERE id = 1; DROP table Article; --
```

Knex('Article').where('id', id); // Knex automatically sanitizes

Here is an example of where want to take advantage of the features of our tools/frameworks for mitigating potential vulnerabilities. This feature isn’t unique to Knex, all frameworks/languages will implement safe substitution in some way. And we want to make sure to use that feature.
We have only scratched the surface of potential vulnerabilities. It is important for us to review the security recommendations for our chosen frameworks and make sure we are following (and staying up-to-date) with best practices. One of the advantages of more comprehensive frameworks (as opposed to lower-level tools like Express) is that they often incorporate these best practices into the implementation by default.

https://owasp.org/www-project-top-ten/