

JSON Web Tokens (JWT)

Course: Java

S1



The background is a dark charcoal gray with a series of parallel diagonal lines running from the top-left to the bottom-right. Overlaid on this are several teal-colored geometric shapes: a large central triangle pointing right, a smaller triangle to its left, and a square to its right. Scattered around these shapes are various white line-art symbols, including a plus sign, a minus sign, a circle with a dot, a circle with a horizontal line, a circle with a vertical line, a circle with a diagonal line, a circle with a cross, a circle with a dot, a circle with a horizontal line, a circle with a vertical line, a circle with a diagonal line, a circle with a cross, a circle with a dot, a circle with a horizontal line, a circle with a vertical line, a circle with a diagonal line, and a circle with a cross.

WELCOME

Learning Outcomes

By the end of today's lesson, you will be able to:

01

Describe how JSON Web Tokens are acquired and used.

02

Acquire a JWT with asynchronous HTTP.

03

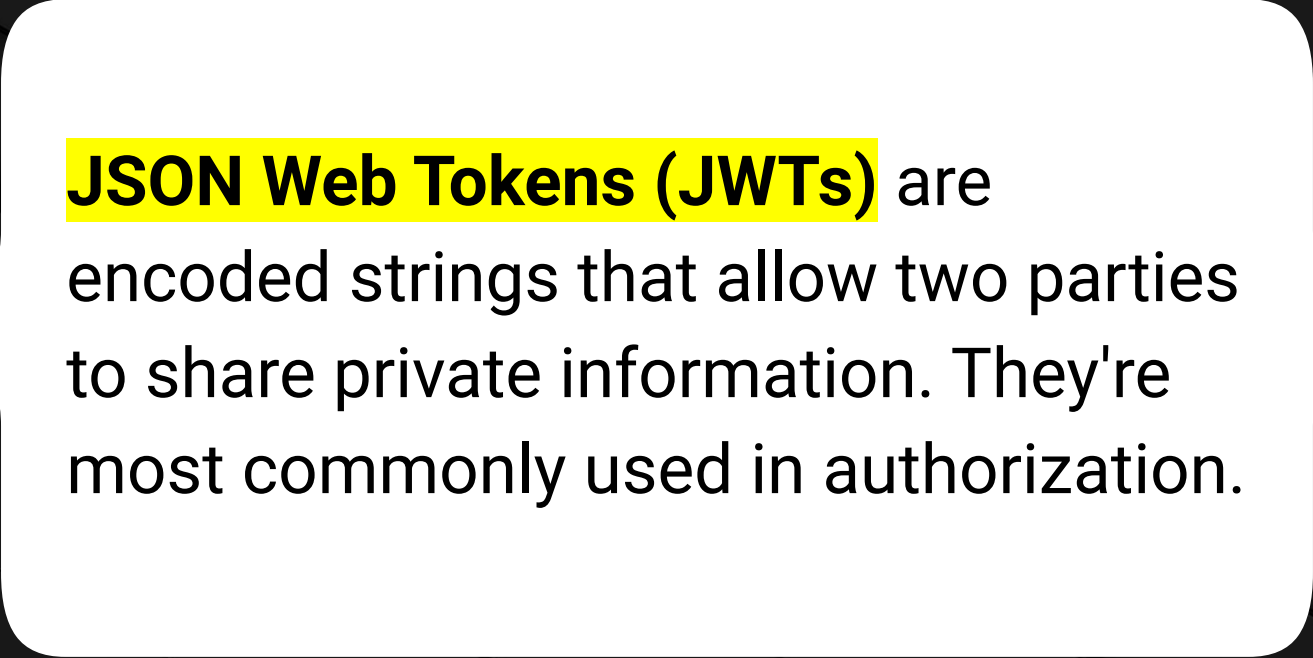
Use a JWT in HTTP requests that require authentication and authorization.

04

Decode a JWT body.

05

Refresh a JWT.



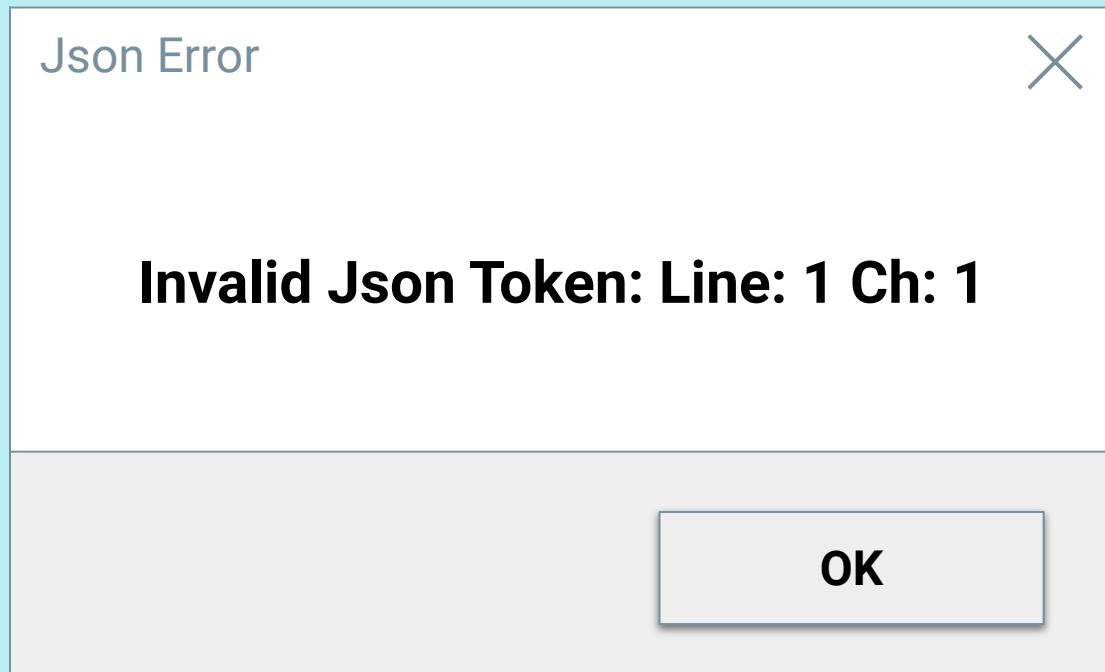
JSON Web Tokens (JWTs) are encoded strings that allow two parties to share private information. They're most commonly used in authorization.



In some cases, JWT-based authorization can be stateless; there's no need to store authorization data in server memory.

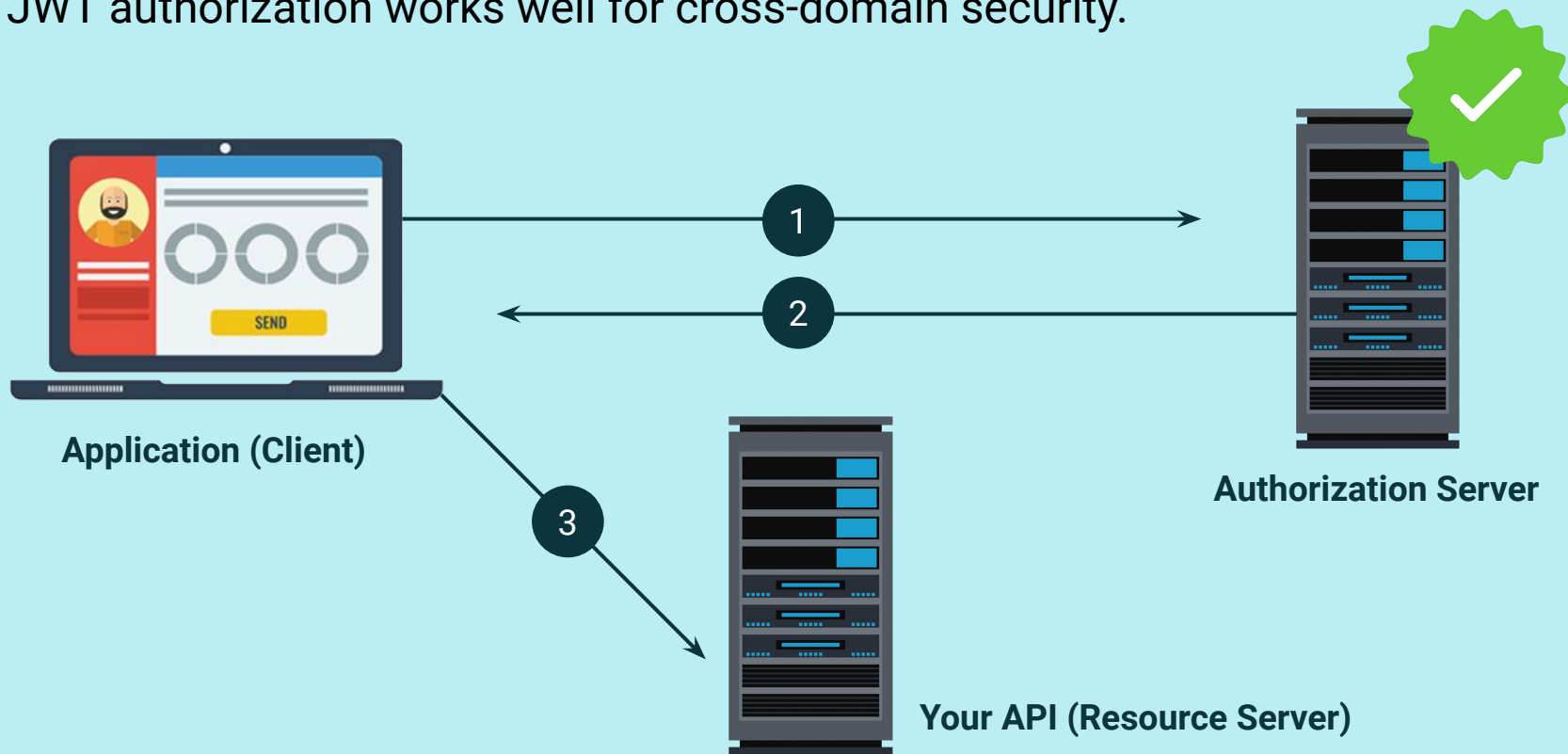
JSON Web Tokens (JWTs)

JWTs are easy to read on both the client and server. They're signed, so any attempt to tamper with the token on the client should result in an **invalid token**.



JSON Web Tokens (JWTs)

JWT authorization works well for cross-domain security.



JSON Web Tokens (JWTs)

There's no need to share cookies across domains.

A **cookie**, sent through the **response header**.

Bob's **session information**, embedded in the **response body**.

```
HTTP/1.1 200 OK
Date: Fri, 13 Mar 2020 12:28:53 GMT
Server: Apache/2.2.14 (Win 32)
Content-Length: 88
Content-Type: text/html
Set-Cookie: cart=Bob
<html><head>Bob's cart<head>...
```



Bob's browser



Some web authorization systems use cookies and session state to manage users and their capabilities.

Web Authorization System

Some web authorization systems use cookies and session state to manage users and their capabilities.

01

The client submits credentials via a form, triggering a same-domain request.

02

The server authenticates the credentials and determines authorizations.

03

The server stores the authorizations in memory, creating a session.

04

The server returns a cookie that identifies the session.

05

From then on, the client submits the session cookie with each request.

06

The server can look up the session and determine if the request is allowed based on permissions.

Web Authorization System

This doesn't work well with independent front-end applications for a few reasons:

01

Independent front-end applications can be hosted on a different domain than their back-end API(s). While it's possible to share cookies cross-domain, not all APIs support it.

02

Without a cookie, there's no friction-free way to share a session identifier.



03

In front-end applications, a session cookie doesn't tell us anything.

We need to know the client's capabilities so that we can decide which UIs to render.



Cookie- or session-based authorization also favors a redirect approach to resource requests.

JSON Web Tokens (JWTs)

If an unauthenticated client makes a request for a protected resource, they're redirected to the login form. On login success, they're redirected to the original protected resource. This doesn't work for front-end applications. They're not required to honor redirection (though they can if they wish).



JWT

JWT is a token-based authorization approach, not a session-based approach.

01

The client submits credentials to an API.

02

The server authenticates and returns a signed, encoded token (string) that contains authorization data.

03

On subsequent requests, the client includes the token with every request.

04

The server verifies the token's integrity and uses it to determine capabilities.

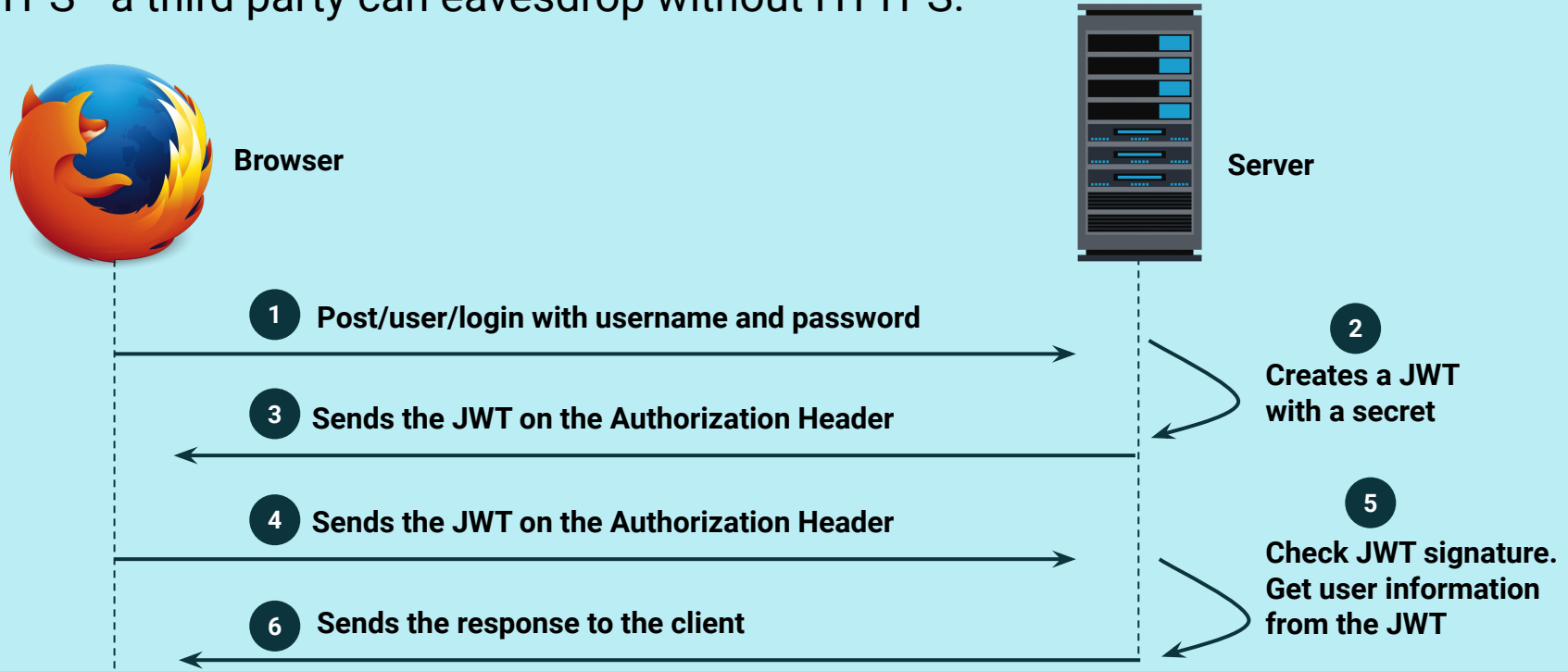
JWT Segments

A typical JWT has three segments: `hhhhhhh.bbbbbbb.ssssss`, where:

hhhhhhh	Is the token header. It contains metadata about the token.
bbbbbbb	Is the token body. It contains data—any data that we decide needs to be shared.
sssssss	Is the signature. It's a cryptographically secure hash of the token. If the signature doesn't match the token, the server knows that someone tampered with it and considers it invalid.

Acquiring a JWT

The authentication call is a typical API call. This should only happen over HTTPS—a third party can eavesdrop without HTTPS.



Acquiring a JWT

01

Authentication is an API call, usually a POST.

02

Add credentials to the request body.

03

On success, we receive JSON in the response body and can read the token.

Acquiring a JWT

Demo code is the only exception:

```
const credentials = {
  username: "user",
  password: "user"
};

const init = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json"
  },
  body: JSON.stringify(credentials)
};

const response = await fetch(`${baseUrl}/authenticate`, init);
if (response.status !== 200) {
  throw "Authentication failed.";
}

const json = await response.json();
const jwt = json["jwt_token"];
```

Using a JWT

Once we have a JWT, we can send it with every request that requires authorization.



Use the `Authorization` header.



The value is “Bearer [jwt value]”.



A JWT doesn't guarantee success. Many API calls depend on roles.

Using a JWT

```
const init = {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Accept": "application/json",
    "Authorization": "Bearer " + this.jwt
  },
  body: JSON.stringify(todo)
};

const response = await fetch(apiUrl, init);
if (response.status === 201) {
  return response.json();
}
return Promise.reject("ToDo was not created.");
```



One JWT advantage is that the token is both an identifier and data. We can bundle any data we want in the body (but be careful with token size).

Role-Based UI

We'll decode roles so that they can be used in conditional UI.

The `sub` (subject) property is also useful. It's usually the user's username.



A JWT is base64 encoded. It's signed but not encrypted.



That means that we can decode the token body to a JSON string.



Then, we can use `JSON.parse` to create a JavaScript object.



JWT bodies can include many things, but it's common for them to include roles.

Role-Based UI

```
const jwt = "eyJhbGciOiJIUzI1NiJ9."
  + "eyJpc3MiOiJ0b2RvLWFwaSIsInN1Y"
  + "iI6InVzZXIiLCJhdXRob3JpdGllcy"
  + "I6IlJPTEVfVWVNFUiIsImV4cCI6MTY"
  + "xNDA1ODc5OX0."
  + "ePv80m0jdSg0vogBQoAGjbFFDq5LM64KbAgzuyvn_ls";

// break up the segments
const segments = jwt.split(".");
// convert from base64
const decoded = atob(segments[1]);
// convert to a live object
const json = JSON.parse(decoded);

console.log(json);
/* {
  iss: "todo-api",
  sub: "user",
  authorities: "ROLE_USER",
  exp: 1614058799
} */

// read the property we want
const role = json.authorities;
```



Conditional UI can be driven by any data i
the token. Where is the best place to store
this data?

Role-Based UI

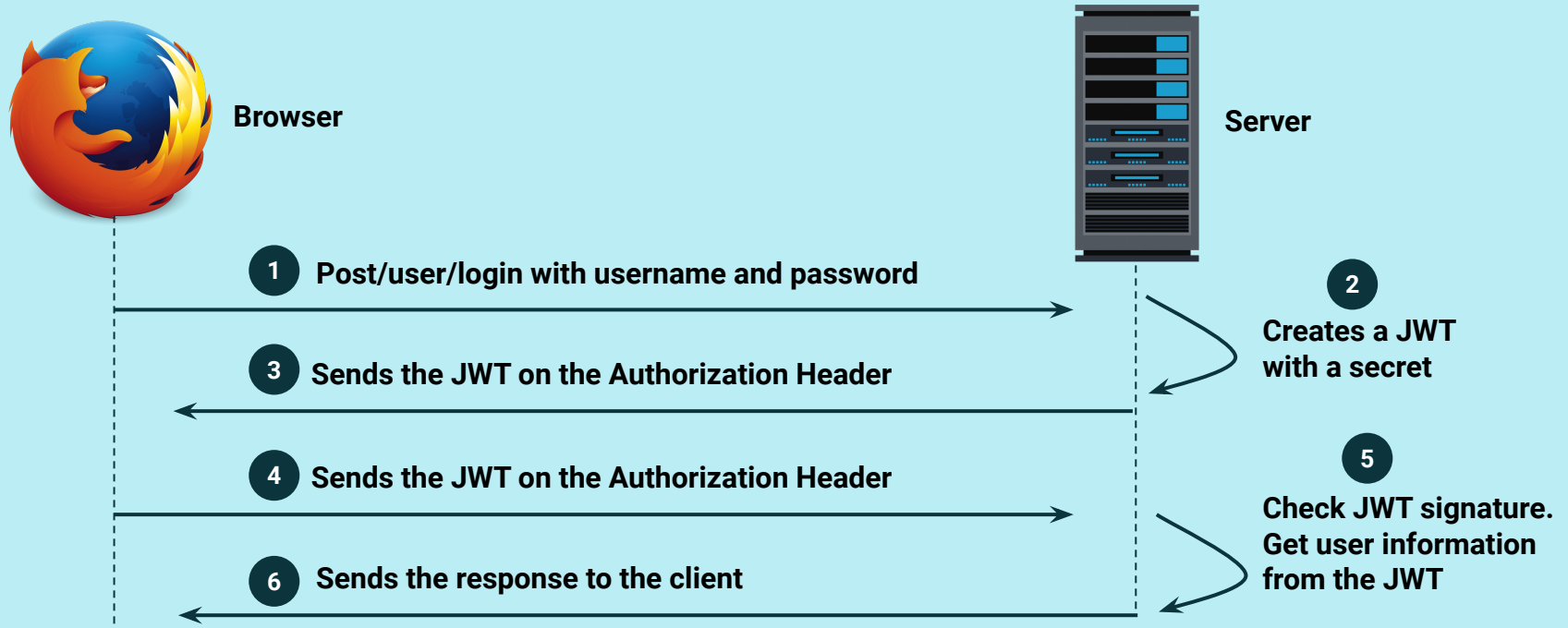
With a role (or authority or capability), we can conditionally show or hide UI elements.

This is not a bad idea to prevent API calls as well, but a secure API should prevent unauthorized actions.

```
<ComponentJsx>
  {role === "CAN_DELETE" && <button>Delete</button>}
  {role === "CAN_EDIT" ? <button>Edit</button>
    : <button>View</button>}
</ComponentJsx>
```

Refresh a Token

In session-based authentication, each request may restart a timer for session timeout. We don't have to worry about expiration as long as we're active.





**With JWT, that's not the case—
there's a fixed expiration.**

**The client is responsible
for restarting the timer.**



**Why don't we explicitly submit
credentials?**

Refresh a Token

We never want to keep a username/password hanging around in memory, and the JWT is a credential. **We use the current JWT to acquire a new JWT.**



Because tokens expire quickly, we need a strategy to “refresh” them. It’s the client’s responsibility.



Most applications use `setTimeout` to schedule a refresh a few seconds before expiration.



The current JWT is sent via `Authorization`. On success, it’s replaced by a new JWT.

Refresh a Token

```
async function login(credentials) {
  // lots of code...
  // on success
  const json = await response.json();
  const jwt = json["jwt_token"];
  setTimeout(() => refreshToken(), INTERVAL);
}

async function refreshToken() {
  const init = {
    method: "POST",
    headers: {
      "Authorization": "Bearer " + this.jwt // old token
    }
  };

  const response = await fetch(`${baseUrl}/refresh_token`, init);
  if (response.status !== 200) {
    throw "Automatic token refresh failed.";
  }

  const json = await response.json();
  const jwt = json["jwt_token"]; // new token
  setTimeout(() => refreshToken(), INTERVAL);
}
```

JWT Variety



There's no one consistent way to manage JWTs.



Search for “React JWT,” and you’ll find lots of options and strong opinions.



For this course, start simple and only add complexity when you need it.



Break



Time to Code



JWT ToDos

Suggested Time:

30 Minutes

JWT ToDos

This updated **ToDo** API is an IntelliJ and Maven project. It's identical to the original API but includes JWT authorization.

Rules:



Everyone, even the unauthenticated, can read **ToDos**.



The **ROLE_USER** and **ROLE_ADMIN** roles can add and update **ToDos**.



The **ROLE_ADMIN** role can delete **ToDos**.

JWT ToDos

There are two built-in, hard-coded users:

username	password	role
user	user	ROLE_USER
admin	admin	ROLE_ADMIN



Recap



How do we acquire a JWT?



How do we use a JWT to gain access to protected actions or resources?



How do we decode a JWT body so that we can use what's inside?



Why does JWT enforce aggressive expirations?



How do we refresh a token before it expires?