Authentication and Authorization for Web Applications

Secure your web app with
JSON Web Tokens

Agenda

- The Demo Application
- JSON Web Token Basics
- JSON Web Tokens and Single Page Apps
- Implementing Authentication with JSON Web Tokens
- Client Side Sessions
- User Information in the Payload
- Protecting Resources
- Protecting Routes

Getting **Started**

https://github.com/chenkie/angular2-user-authentication

BONUS

https://github.com/chenkie/angular1-user-authentication

https://github.com/chenkie/react-user-authentication

https://github.com/chenkie/user-authentication-api

ZEIT

http://bit.ly/fem-jwt-api

FemAuthAngular2 Ryan

localhost:4200/profile

Angular 2 Authentication Instructors Profile Log Out

Profile

chenkie

ryan@elevatedigital.io

Payload


```
{
  "sub": "5816530f1eaa5d4db4e2cf08",
  "username": "chenkie",
  "email": "ryan@elevatedigital.io",
  "gravatar": "https://www.gravatar.com/avatar/e226e2760fa13be8110b118fef90efb3",
  "iat": 1478483284,
  "exp": 1478486884
}
```

The Demo API

- Simple REST API that allows users to sign up and log in
- When a user is authenticated, a JSON Web Token is returned in the response
- The API has a resource called `instructors` which is a listing of several Front End Masters instructors

The Demo Front End App

- The demo app is provided in three varieties: Angular 1.5, Angular 2, and React
- The front end app that we'll work on will allow users to sign up or log in, view their profile, request a listing of instructors and add new instructors
- To view the profile area and list of instructors, users will need to be logged in
- To add new instructors, users will need to be an administrator

Challenges

- Run the finished app (in the framework of your choosing)
- Access the API and view the documentation for its endpoints in the `readme`: https://github.com/chenkie/user-authentication-api
- If you want to, clone and run the API locally (make sure to read the instructions)
- Optional: create a user for yourself at the `/api/users` endpoint from Postman

JSON Web Token Basics

What is a JSON Web Token (JWT)?

- An open standard: RFC 7519
- A method for transferring claims (assertions) between two parties securely through a JSON payload
- A digitally signed and compact, self-contained package
- A great mechanism for **stateless** authentication

Basic JWT

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Signature

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  <secret>
)
```

JWT Header

- JSON object that describes the token
- At a minimum it should include the token type and signing algorithm
- The signing algorithm is necessary for the token to be verified
- Commonly tokens are signed with HS256 (symmetric) or RS256 (asymmetric)
- Header example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JWT Payload

- JSON object which contains any claims (assertions) about the entity for which it was issued
- The JWT standard describes a set of reserved claims
  - `iss, sub, aud, exp, nbf, iat, jti`
- The payload can also contain any arbitrary claims defined at will

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

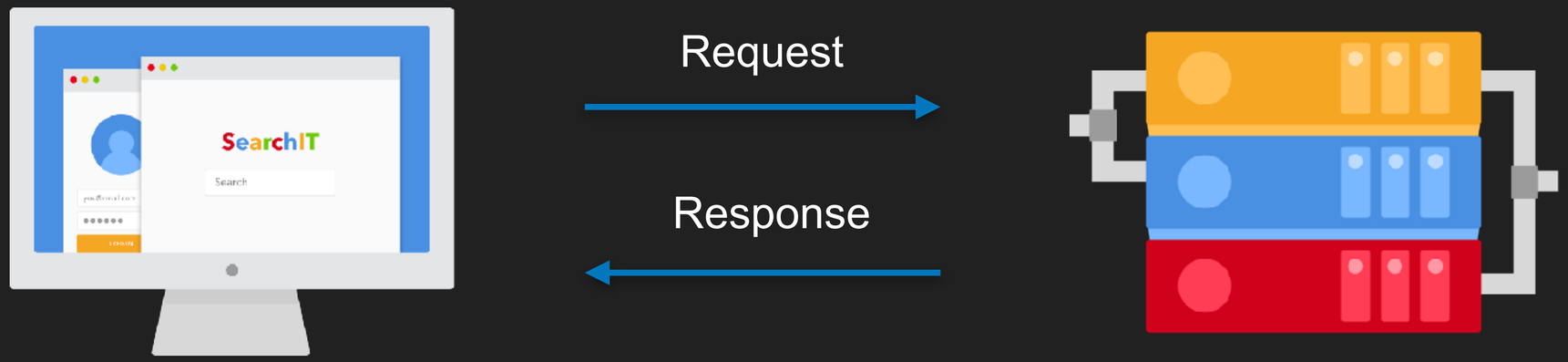
JWT Signature

- JSON object produced by Base64 URL encoding the header and payload and then running them through a hashing algorithm with a secret key
- The signature is used as a means to digitally sign the token so that its validity can be verified later
- If anything in the header or payload is modified, the token will be invalidated

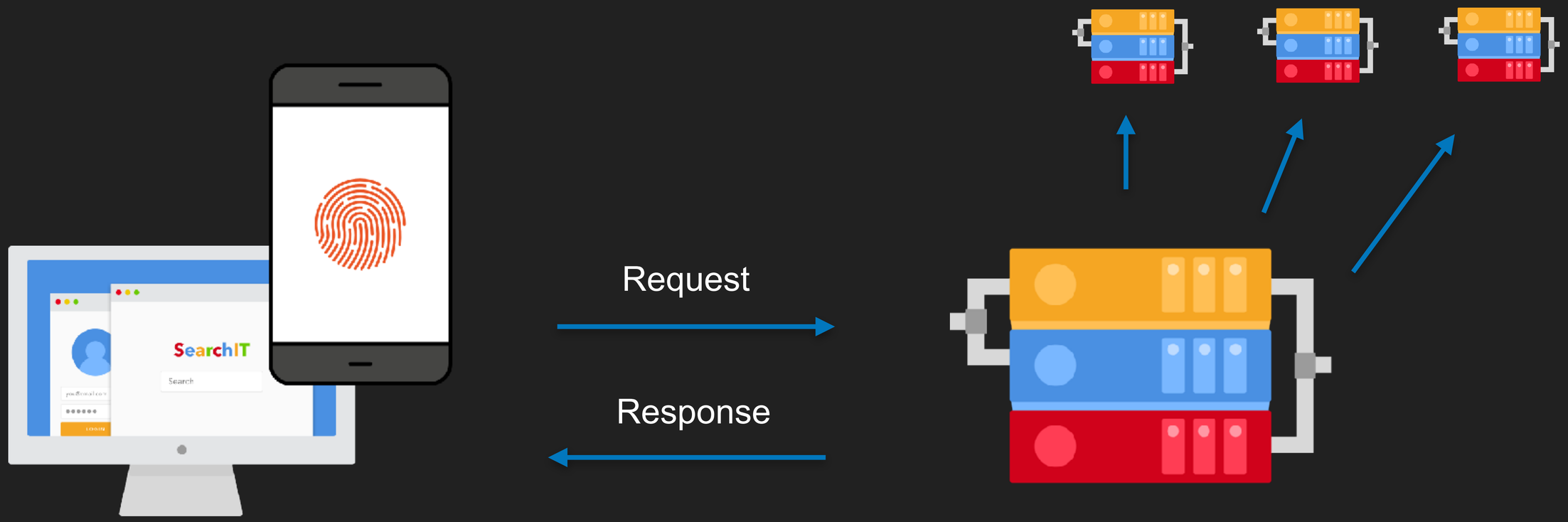
```
HMACSHA256(
  base64UrlEncode(header) +
"." +
  base64UrlEncode(payload),
  <secret>
)
```

JSON Web Tokens and **Single Page Apps**

Traditional Client-Server Interactions Were Straightforward

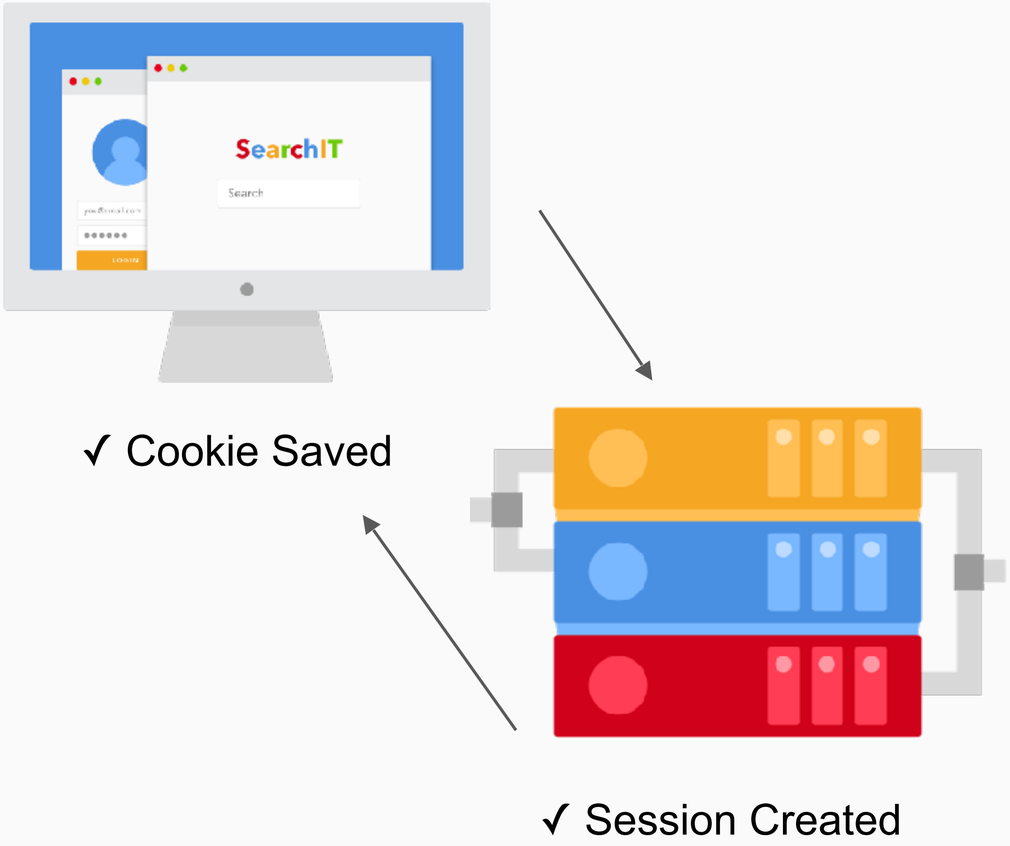


Modern Interactions are More Complex

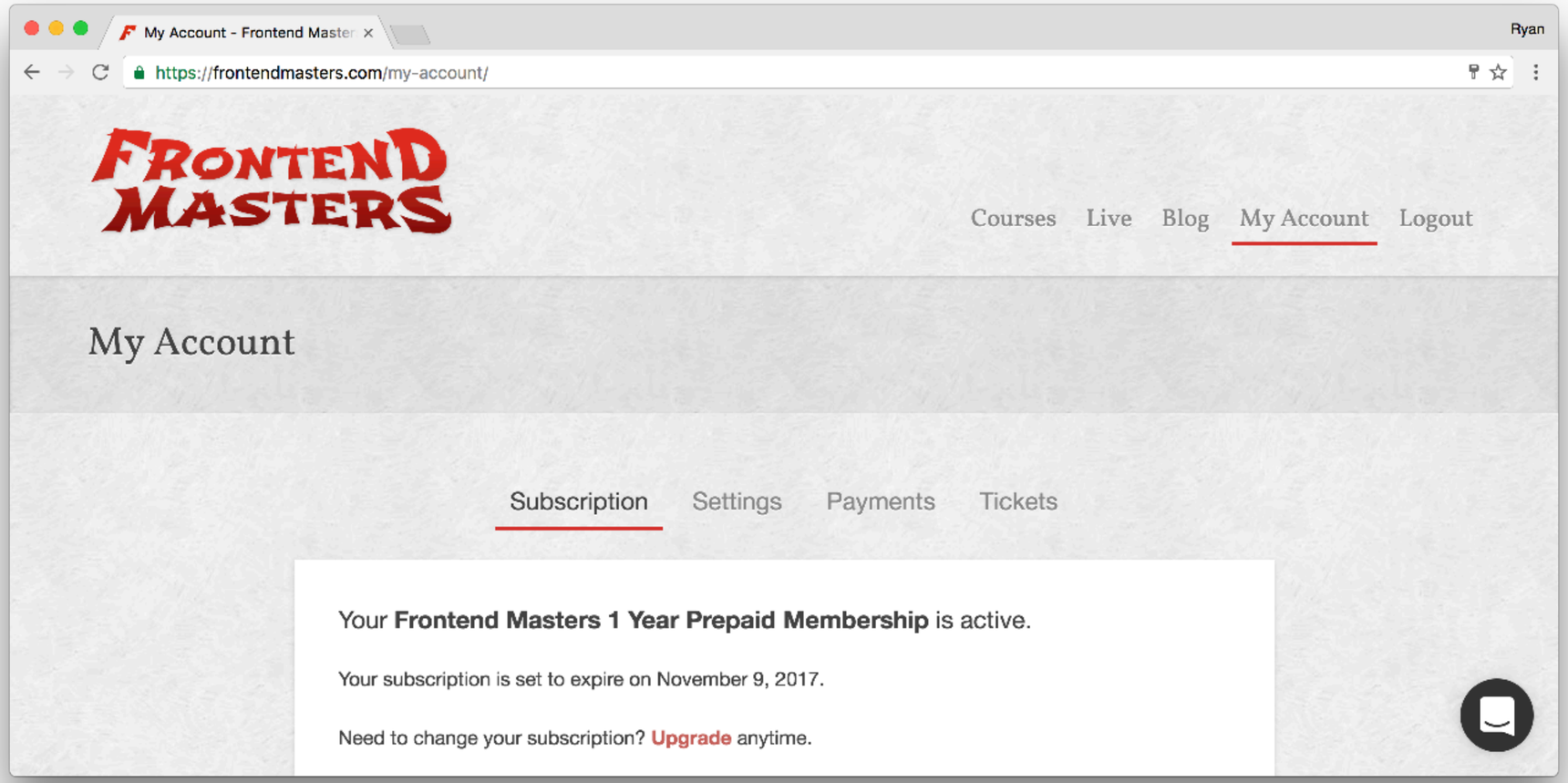


Traditional Authentication

- The user submits their credentials which are checked against a database
- If everything is good, a session is created for them on the server, and a cookie with a `session_id` is sent back to the browser
- The cookie is sent back to the server on all subsequent requests and is verified against the session



Traditional Authentication Example



Downsides to Cookie/Session Auth

Let’s explore some problems with traditional authentication in SPAs

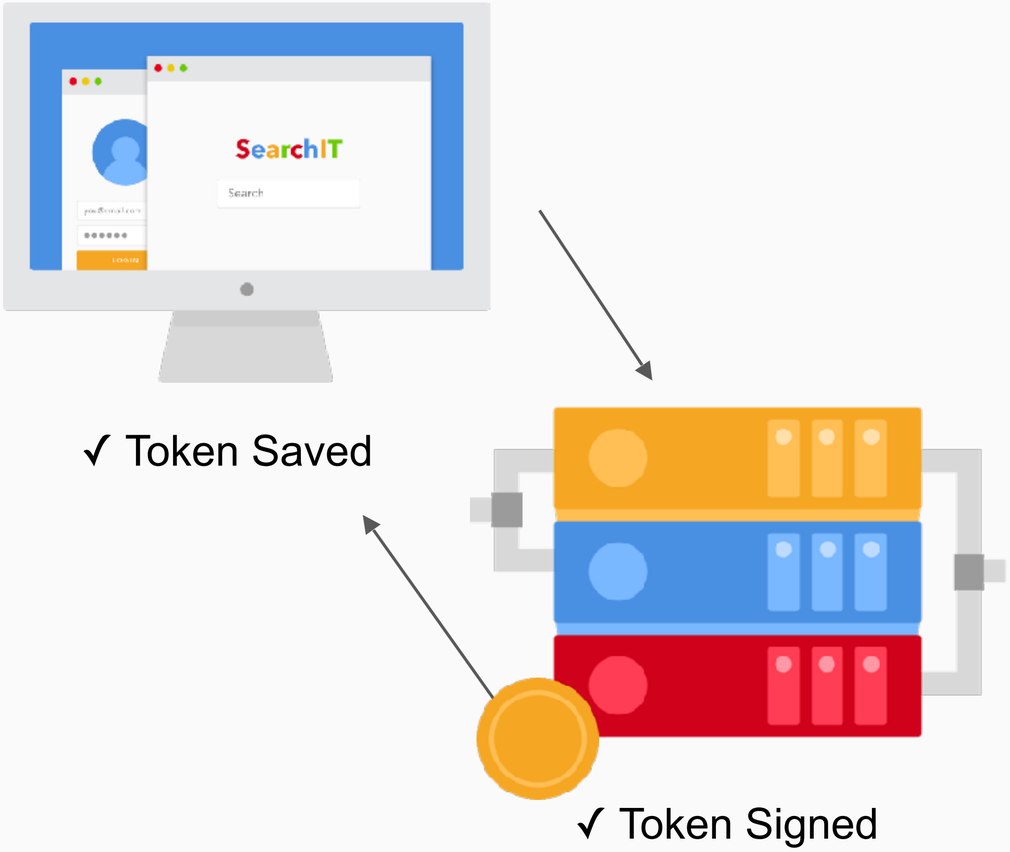
- Since the SPA doesn’t refresh, how does it know that the user is logged in?
  - Traditional apps construct views on the backend which is where auth can be checked
- How do SPAs get their data? Generally a REST API
  - REST APIs should be **stateless** and traditional authentication is **stateful**

Downsides to Cookie/Session Auth

- Modern architecture relies on micro services and downstream servers but cookies don’t flow downstream
  - Can’t communicate easily between multiple servers with traditional auth
- Access control requires database queries
  - General chattiness on the backend
- Doesn’t scale well and can become memory-intensive
- In traditional authentication, the server does the heavy lifting

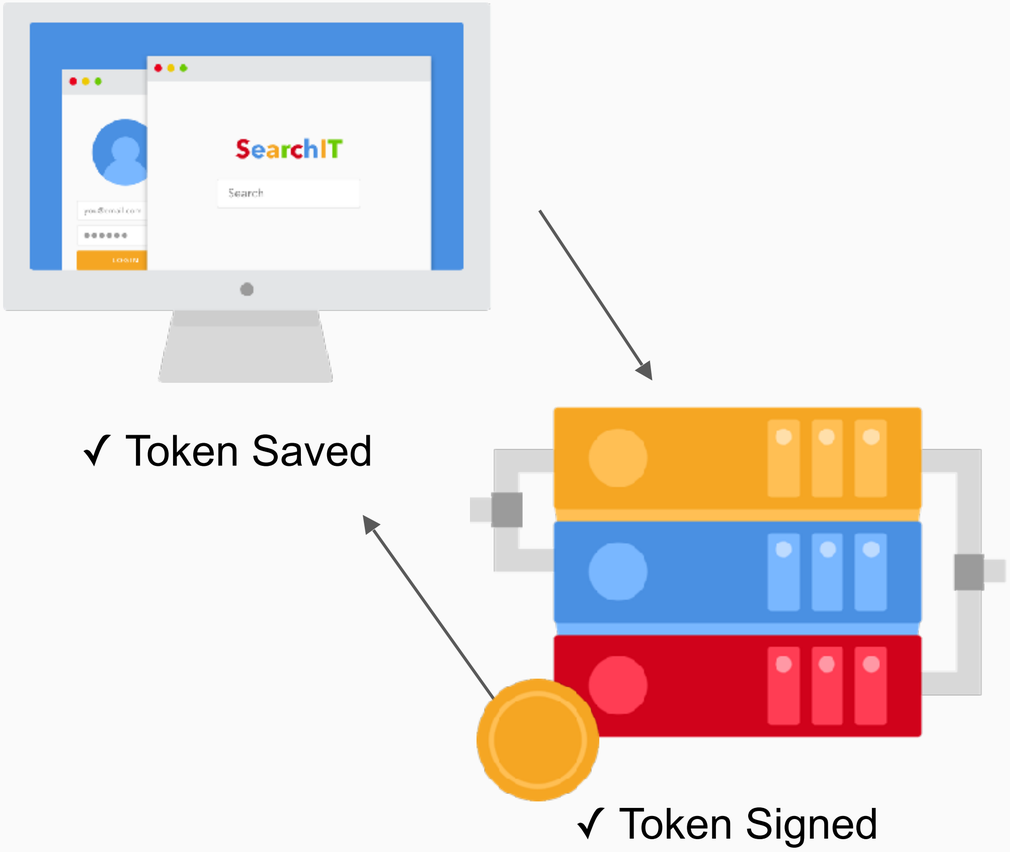
JWT Authentication

- The user submits their credentials which are checked against a database
- If everything is good, a token is signed and returned to the client in the response
- The token is saved on the client, usually in web storage or in a cookie
- The token is sent as an `Authorization` header on every HTTP request



JWT Authentication

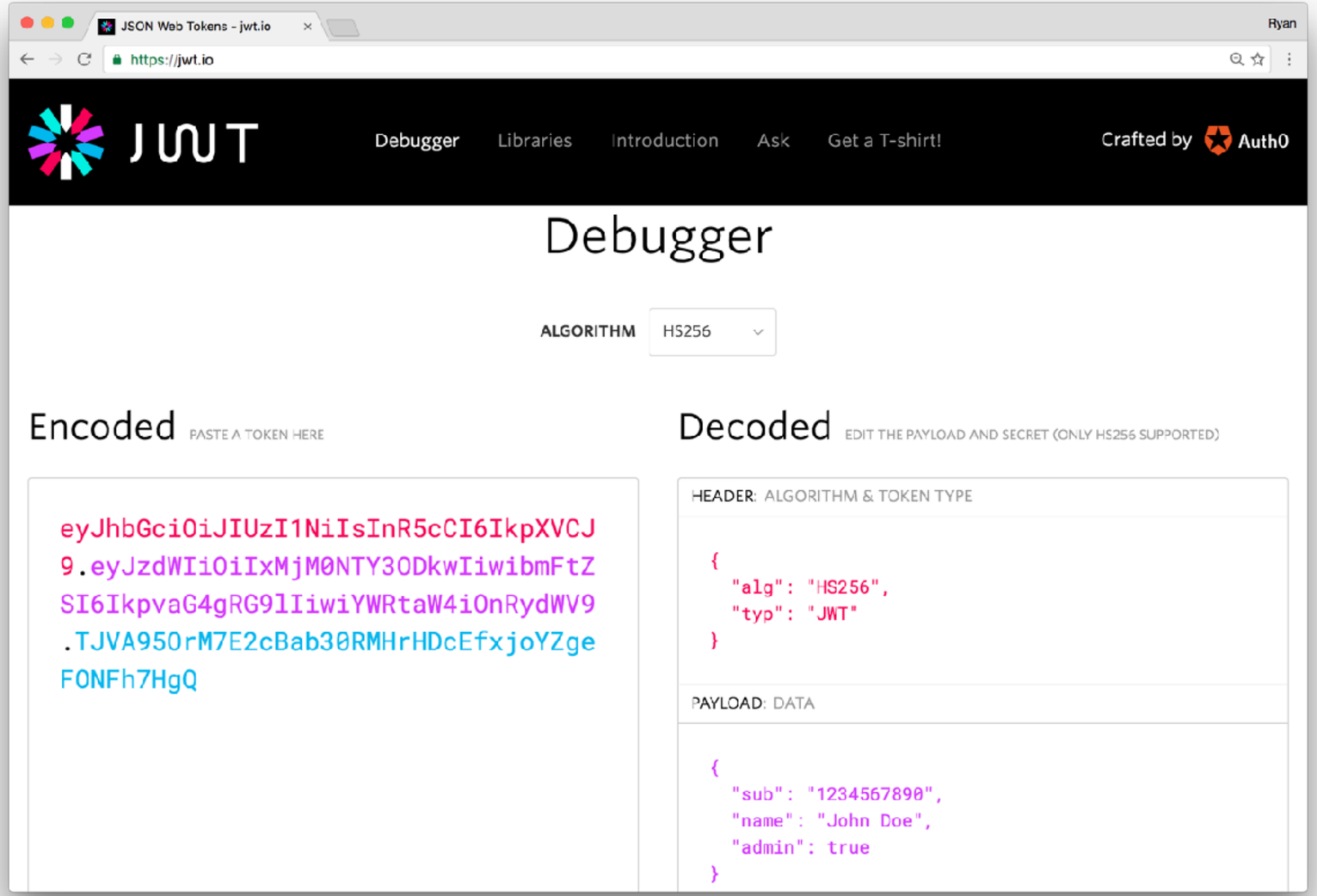
- When the request is received on the backend, the JWT is verified against the secret that only the server knows
- The payload is checked to route the request based on the JWT's claims (usually with middleware)
- If the JWT is valid, the requested resource is returned
- If it is invalid, a 401 is returned



How Does This Help?

- The SPA no longer relies on the backend to tell it whether the user is authenticated
- The backend can receive requests from multiple clients and the backend only cares that the token is valid
- Requests can flow to downstream servers if necessary
- The client tells the backend what is permissible instead of asking
  - No need for user access lookups

Quick Exercise: Try it out with jwt.io



Implementing **Authentication** with JSON Web Tokens

How Does the SPA Get a JWT?

- The user submits their credentials
- If the credentials are valid, a JWT is signed and returned in the response
- A secret key (only known by the server) is used to sign the JWT

Exploring the
JWT Signing Code

JWT Storage

- Once the JWT comes back, it needs to be stored somewhere in the user’s browser
- Storing it in memory isn’t great because it will be lost when the page is refreshed
- JWTs are typically stored in browser storage (local storage or session storage) or in HTTP-only cookies

```
finishAuthentication(token): void {
  localStorage.setItem('token', token)
  this.router.navigate(['profile']);
}

logout(): void {
  localStorage.removeItem('token');
}
```

Storing the JWT

Basic
Authentication
Sequence

```
export class LoginComponent {
  errorMessage: string;

  constructor(private auth: AuthService) { }

  onLoginSubmit(credentials) {
    this.auth.login(credentials)
      .map(res => res.json())
      .subscribe(
        response => this.auth.finishAuthentication(response.token),
        error => this.errorMessage = error.json().message
      );
  }

  onSignupSubmit(credentials) {
    this.auth.signup(credentials)
      .map(res => res.json())
      .subscribe(
        response => this.auth.finishAuthentication(response.token),
        error => this.errorMessage = error.json().message
      );
  }
}
```

Login Component

```
onLoginSubmit(credentials) {
  this.auth.login(credentials)
    .map(res => res.json())
    .subscribe(
      response => this.auth.finishAuthentication(response.token),
      error => this.errorMessage = error.json().message
    );
}
```

Login

```
onSignupSubmit(credentials) {
  this.auth.signup(credentials)
    .map(res => res.json())
    .subscribe(
      response => this.auth.finishAuthentication(response.token),
      error => this.errorMessage = error.json().message
    );
}
```

Signup

```
@Injectable()
export class AuthService {
  constructor(private http: Http, private router: Router) { }

  login(credentials): Observable<Response> {
    return this.http.post(`${API_URL}/users/authenticate`, credentials);
  }

  signup(credentials): Observable<Response> {
    return this.http.post(`${API_URL}/users`, credentials);
  }

  finishAuthentication(token): void {
    localStorage.setItem('token', token)
    this.router.navigate(['profile']);
  }

  logout(): void {
    localStorage.removeItem('token');
  }
}
```

AuthService

```
@Injectable()
export class AuthService {
  constructor(private http: Http, private router: Router) { }

  login(credentials): Observable<Response> {
    return this.http.post(`${API_URL}/users/authenticate`, credentials);
  }

  signup(credentials): Observable<Response> {
    return this.http.post(`${API_URL}/users`, credentials);
  }

  finishAuthentication(token): void {
    localStorage.setItem('token', token)
    this.router.navigate(['profile']);
  }

  logout(): void {
    localStorage.removeItem('token');
  }
}
```

AuthService

Challenges

- Checkout branch **01-implementing-auth-starter**
- Create a screen for `login` and `signup`
- Make a POST request with the user's credentials
- Store the JWT that comes back in local storage
- Create a logout method which removes the token from local storage
- Provide buttons for **Log In** and **Log Out** in the home view and the toolbar

Client Side **Sessions**

About Sessions

- What is a session?
  - In general terms, a session is a way to preserve a desired state

- What is a server-side session?
  - It’s a piece of data stored in memory on the server (or in a database) that tracks information
  - For authentication, this generally means an identifier for the user
  - Ultimately it is used to make a determination about the user’s authentication status
  - Keeping server-side sessions in this way is stateful

About Sessions

- What is a client-side session?
  - SPAs require a way to know whether a user is authenticated or not
  - Can’t be done in a traditional manner because the SPA is largely decoupled from the backend
  - JWT is a stateless authentication mechanism, which means no user session exists on the server anyway

Client Sessions

- How can we have client sessions using a stateless authentication mechanism?
  - The best indication we can go by is whether or not the user's JWT has expired

- Rationale
  - If the JWT has expired, it can't be used to access protected resources
  - Since authentication in this scenario is broadly concerned with protecting resources, it can be used as an indicator of authentication state

Client Sessions

- When the user logs in, provide an application-wide flag to indicate the user is logged in
- At any point in the application's lifecycle, the token's `exp` value can be checked against the current time
- If the token expires, change the flag to indicate the user is logged-out
- The check is commonly done when a route change occurs
  - If the token is expired, redirect the user to the login route
  - Toggle appropriate markup for the user being logged out

```
export class AuthService {
  constructor(private http: Http, private router: Router) { }

  // ...

  isAuthenticated(): boolean {
    return tokenNotExpired('token');
  }

  // ...
}
```

isAuthenticated

```
<div *ngIf="isAuthenticated">
  <p> Welcome, {{ name }}!
    View your <a href="/profile">profile</a> or
    <a routerLink="/logout">log out</a>.
  </p>
</div>

<div *ngIf="!isAuthenticated">
  <p> Welcome! Please <a routerLink="/login">log in</a>.</p>
</div>
```

```
export class AuthService {
  constructor(private http: Http, private router: Router) { }

  // ...

  isAdmin(): boolean {
    return jwtDecode(this.getToken()).scope === 'admin';
  }

  getUseRole(): string {
    return jwtDecode(this.getToken()).scope;
  }

  getToken(): string {
    return localStorage.getItem('token');
  }
}
```

AuthService.isAdmin

```
<button
  class="btn btn-primary"
  *ngIf="auth.isAuthenticated() && auth.isAdmin()"
  routerLink="/instructor/new">
  <i class="glyphicon glyphicon-plus"></i> Add Instructor
</button>
```

auth.isAdmin

```
<a class="btn btn-primary btn-lg"
  routerLink="/login"
  *ngIf="!auth.isAuthenticated()"
  role="button">Log In</a>

<a class="btn btn-primary btn-lg"
  (click)="auth.logout()"
  *ngIf="auth.isAuthenticated()"
  role="button">Log Out</a>
```

isAuthenticated

Challenges

- Checkout branch **02-client-sessions-starter**
- Implement a function which uses the JWT's expiry time to check whether the user is authenticated
  - Hint: there are libraries to help!
- Conditionally hide and show elements based on authentication state

User Information in the Payload

Payload Refresher

- The JWT’s payload contains claims which are assertions about a subject
- We can assert various things about a user
  - Name
  - Email
  - Picture

```
export class ProfileComponent implements OnInit {

  profile: Profile;
  payload: Object;

  constructor(private auth: AuthService) {
  }

  ngOnInit() {
    this.profile = jwtDecode(this.auth.getToken());
    this.payload = jwtDecode(this.auth.getToken());
    this.profile.gravatar = `${this.profile.gravatar}?s=200`;
  }
}
```

ProfileComponent

Payload Best Practices

- It might be tempting to put a whole profile object in the payload, but we shouldn’t do this
- It’s important to keep the JWT small because it is sent over the wire on all requests
- Since the JWT is decodable, we want to keep sensitive information out

Payload Best Practices

- What should be in the payload?
  - Basic user information
  - Nothing secret or sensitive

- Consider providing a separate endpoint which retrieves a user profile object if you need a lot of profile data

Challenges

- Checkout branch **03-user-profile-starter**
- Read the user's profile out of the JWT payload
  - Hint: there are libraries to help!
- Display the user's details in a `profile` view

Protecting **Resources**

Protecting Resources

- The point of adding authentication to an app is to restrict resource access to users who have proven they are allowed to access those resources

- Different levels of access
  - **Publicly accessible** — data is open to anyone
  - **Limited to authenticated users** — data is open to anyone who is logged in
  - **Limited to only one authenticated user** — data is open to only the user who is logged in
  - **Limited to a subset of authenticated users** — data is open to anyone of a particular privilege

Protecting Resources

- How do JWTs help us to protect resources?
  - We can create endpoints for our resources that require an authentication check
  - To pass the check, a valid JWT must be present
  - When making HTTP requests, we can send the JWT as an `Authorization` header
  - The header is read at the API and if it’s valid, the resource is accessible

Exploring the
JWT Middleware Code

Making Authenticated Requests

- Sending authenticated requests requires retrieving the JWT from storage and attaching it as an `Authorization` header.
- Some common ways this is implemented include:
  - Explicitly on a per-request basis
  - Globally on all requests
  - Only requests of a certain kind (method and resource type)
- Storing JWT in a Cookie means that it goes to the server on every request

```
this.http.get(API_URL, headers: { 'Authorization': 'Bearer ' + token })
  .map(res => res.json())
  .subscribe(data => console.log(data));
```

Auth Headers

Let’s get **lazy!**

```
@NgModule({
  declarations: [],
  imports: [],
  providers: [
    AuthService,
    AuthGuard,
    RoleGuard,
    provideAuth({
      tokenGetter: () => { return localStorage.getItem('token') }
    }),
    InstructorService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

provideAuth

```
export declare class AuthHttp {
    private http;
    private defOpts;
    private config;
    tokenStream: Observable<string>;
    constructor(options: AuthConfig, http: Http, defOpts?: RequestOptions);
    private mergeOptions(providedOpts, defaultOpts?);
    private requestHelper(requestArgs, additionalOptions?);
    private requestWithToken(req, token);
    setGlobalHeaders(headers: Array<Object>, request: Request | RequestOptionsArgs): void;
    request(url: string | Request, options?: RequestOptionsArgs): Observable<Response>;
    get(url: string, options?: RequestOptionsArgs): Observable<Response>;
    post(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>;
    put(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>;
    delete(url: string, options?: RequestOptionsArgs): Observable<Response>;
    patch(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>;
    head(url: string, options?: RequestOptionsArgs): Observable<Response>;
    options(url: string, options?: RequestOptionsArgs): Observable<Response>;
}
```

AuthHttp

```
export class InstructorService {

  constructor(private http: Http, private authHttp: AuthHttp) { }

  public getInstructors(): Observable<Response> {
    return this.authHttp.get(`${API_URL}/instructors`);
  }

  public addInstructor(data): Observable<Response> {
    return this.authHttp.post(`${API_URL}/instructors`, data);
  }
}
```

this.authHttp

Authorization Schemes

- There are various schemes registered for the `Authorization` header
- The Bearer scheme is borrowed from OAuth 2.0
- Other common schemes include `Basic` and `Digest`

Challenges

- Checkout branch **04-protecting-resources-starter**
- Set up the application to send the JWT in an `Authorization` header when needed
- Make a `GET` request to the API for the `instructors` resource and display the list in the app
- Make a `POST` request to the API to add a new `instructor`
  - Hint: there are libraries to help!

Protecting **Routes**

Client-Side Considerations

- Server resources are limited to only requests which have a valid JWT
- But what about limiting access on the client side?
- Some client side considerations:
  - Users should only be able to navigate to protected routes if they are authenticated
  - If a route requires a certain access level, users should only be able to navigate there if they have the appropriate scope
  - Certain UI elements should only be rendered if the above conditions are met

Protecting Routes

- However, protecting client side routes and UI elements has a big problem: it's easy to forge
  - The user can modify the `exp` time or `scope` in their own JWT
  - We can't verify the signature of the JWT on the client side because the secret can never leave the server

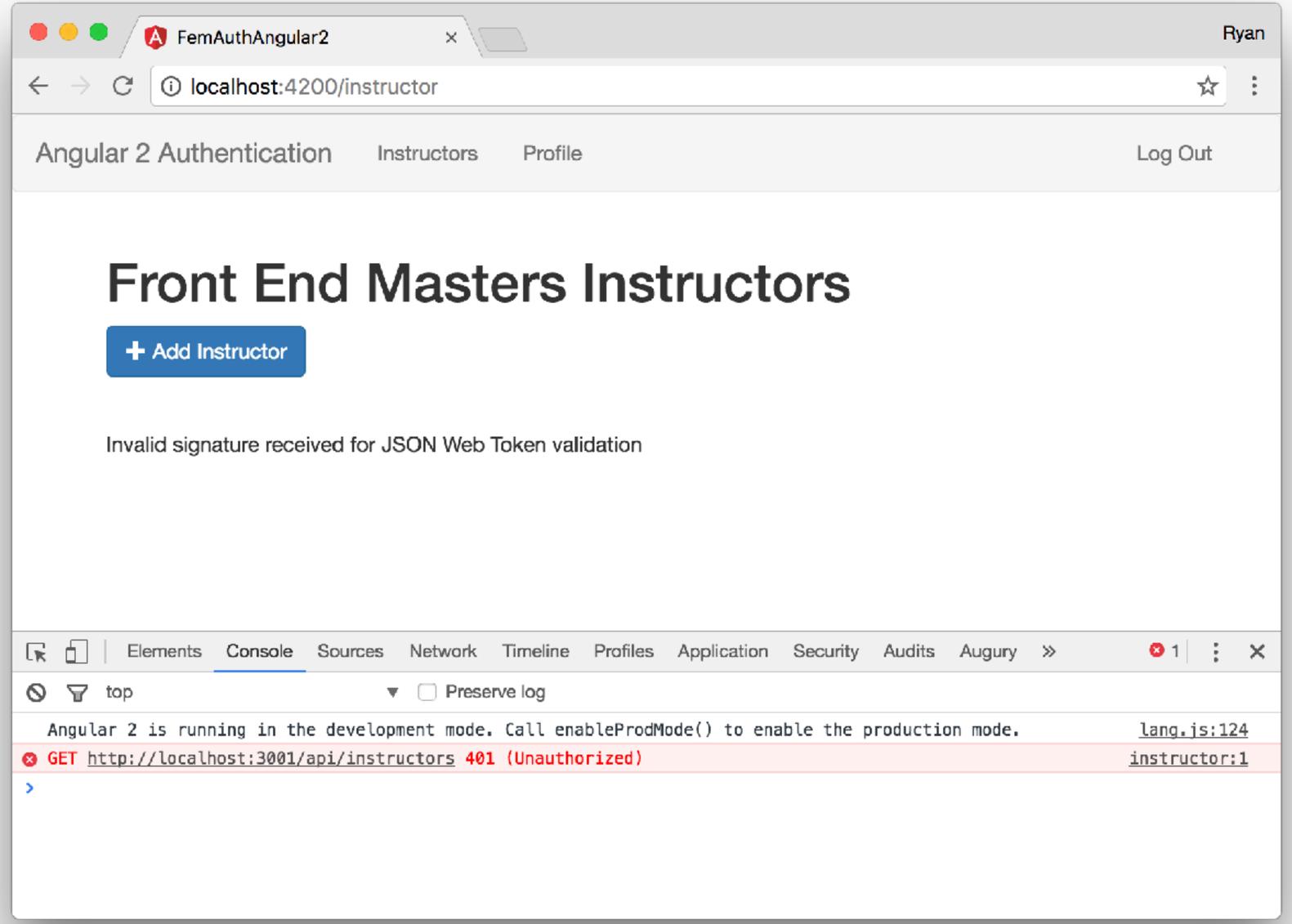
Protecting Routes

- But does it matter?
- In the end, protected resources should remain on the server anyway
  - Anything in the client-side is easily visible by anyone who can use dev tools
- If a savvy user manages to hack their way to a protected route (either by modifying their JWT or otherwise tampering with the code), they won't be able to get the resources from the server

Scenario: Savvy User Modifies the JWT

- We use JavaScript to limit a route to authenticated users who also have a `scope` of `admin`
- A savvy user who only has a `scope` of `user` decides to modify their JWT in the jwt.io debugger
- They gain access to the route which is populated by resources from the server
- What happens?

Request with Invalid JWT



How are Client-Side Routes Protected?

- When a route transition starts, the `exp` time in the JWT payload is checked
  - If the JWT is expired, the transition is disallowed
- If a route requires a certain access level, the `scope` in the JWT is checked when the route transition starts
  - If the JWT doesn't include the desired `scope`, the transition is disallowed

Details Differ by Framework

- Many frameworks have their implementations for controlling route access
  - Angular 1.x - router events (`$routeChangeStart`, `$stateChangeStart`)
  - Angular 2 - route guards which implement a `CanActivate` hook
  - React (React Router) - `onEnter` event

```
export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  { path: 'home', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'instructor', component: InstructorComponent, canActivate: [AuthGuard] },
  { path: 'instructor/new', component: NewInstructorComponent, canActivate: [RoleGuard] },
  { path: 'profile', component: ProfileComponent }
];
```

Guarded Routes

```
export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  { path: 'home', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'instructor', component: InstructorComponent, canActivate: [AuthGuard] },
  { path: 'instructor/new', component: NewInstructorComponent, canActivate: [RoleGuard] },
  { path: 'profile', component: ProfileComponent }
];
```

Guarded Routes

```
export interface CanActivate {
    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
                                Observable<boolean> | Promise<boolean> | boolean;
}
```

CanActivate

```
@Injectable()
export class AuthGuard implements CanActivate {

  constructor(private auth: AuthService, private router: Router) {}

  canActivate() {
    if (this.auth.isAuthenticated()) {
      return true;
    } else {
      this.router.navigate(['login']);
    }
  }
}
```

AuthGuard

```
@Injectable()
export class RoleGuard implements CanActivate {

  constructor(private auth: AuthService, private router: Router) {}

  canActivate() {
    if (this.auth.isAuthenticated() && this.auth.isAdmin()) {
      return true;
    } else {
      this.router.navigate(['login']);
    }
  }
}
```

RoleGuard

Challenges

- Checkout branch **05-protecting-routes-starter**
- For the `instructor` route, check that the user's JWT is unexpired before the route transition happens
- For the `instructor/new` route, check that the user's JWT is unexpired and that they have a `scope` of `admin`
- Hide the New Instructor button if the user isn't an `admin`

Further Reading & **Wrap-Up**

Important Considerations

- Nothing is 100% secure and JWTs are no exception
- Common attack vectors:
  - XSS (if using local storage)
  - CSRF (if using cookies)
  - MITM attacks
- Always serve your app and API over HTTPS
- Always escape user input and put CSRF protection in place if necessary

Important Considerations

- JWT describes how computers can communicate securely between one another but it doesn’t say anything about how suitable your own implementation might be
- It’s up to you to make a determination about whether your implementation is secure
- OAuth 2.0 and OIDC standardize authentication and authorization
  - While complex, they may be the best solution in some scenarios

Further Reading

- Auth0 Blog: https://auth0.com/blog
- JWT Standard (RFC 7519): https://tools.ietf.org/html/rfc7519
- OAuth 2.0 Framework (RFC 6749): https://tools.ietf.org/html/rfc6749
- OpenID Connect: https://openid.net/connect/

@ryanchenkie
@simpulton

Thanks!