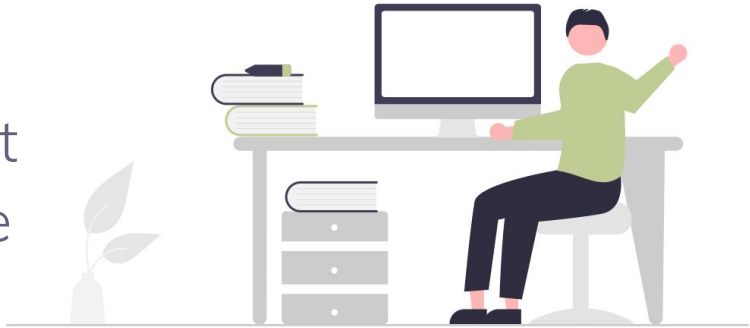
Staying Safe and Secure with Angular

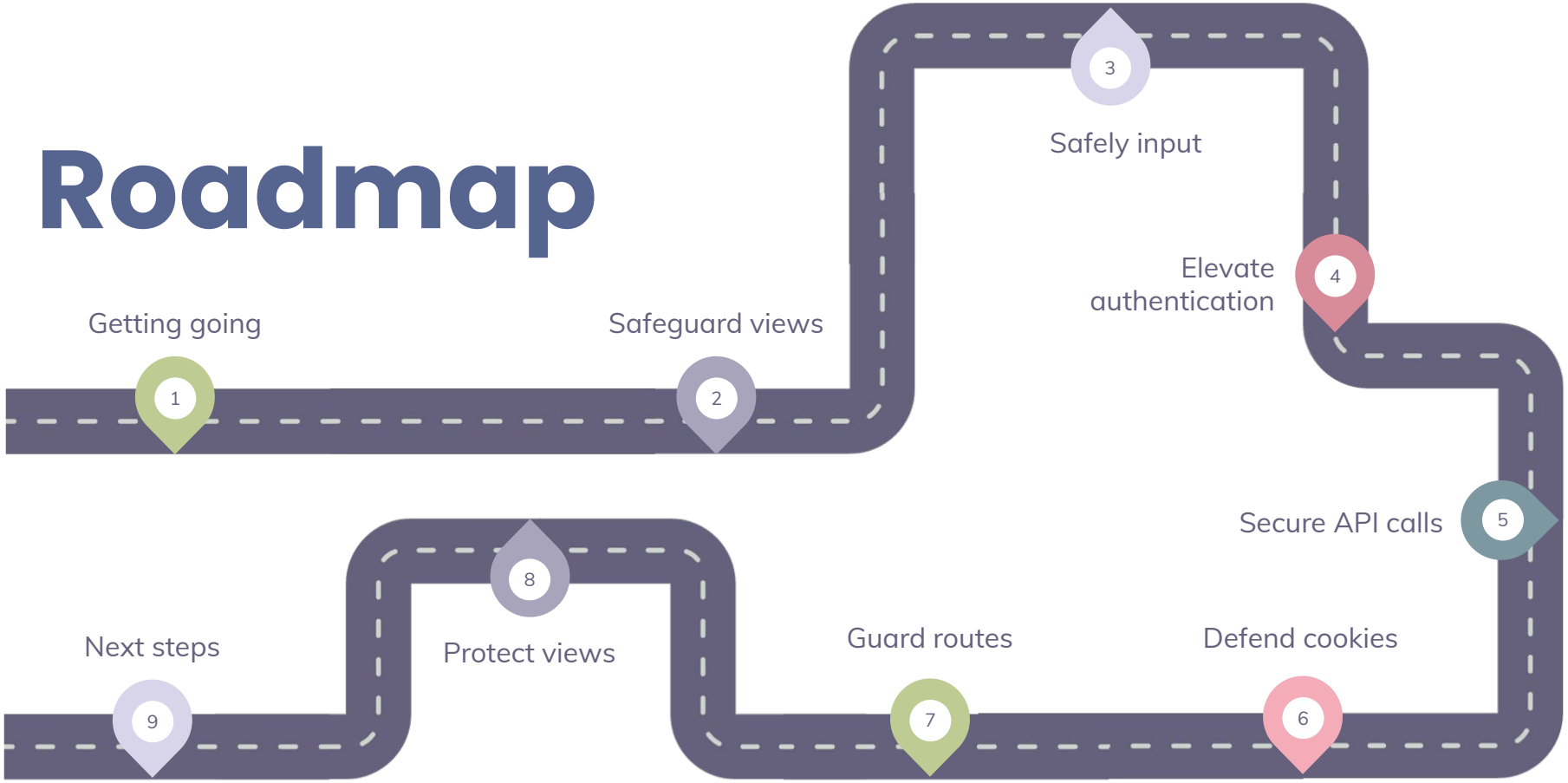
Wifi

You will need

Node v18, Git, IDE with TypeScript auto imports & Angular Language Service, Angular CLI



Roadmap


1
Getting going
2
Safeguard views
3
Safely input
4
Elevate authentication
5
Secure API calls
6
Defend cookies
7
Guard routes
8
Protect views
9
Next steps

It’s just Angular

We’ll examine the Angular concepts you already know, but with an eye for secure coding

@for module of modules

1. **Understand attack vectors**
   Cover attack vector for common tasks and simulate the attack locally
2. **Refactor or add features**
   Refactor the starting application to use secure coding techniques, and add new features securely
3. **Review concepts and cover takeaways**
   Review what we learned and did, discuss actionable takeaways

Hello!

Alisa Duncan

Dev Advocate @ Okta | Angular GDE

@AlisaDuncan

#secureWithAngular

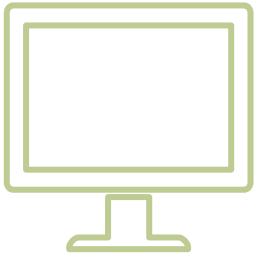
Let's all be friends

Introduce yourself to your neighbor! They will be your buddy today.

1.
Getting going

Intro to OWASP
Run starting application

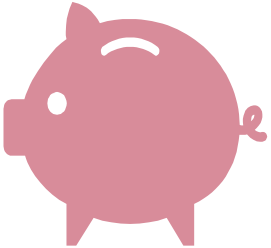
Web vulnerabilities can cause risks to your assets



Application

Data

Reputation

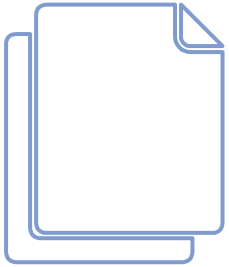


Bottom line

Web vulnerabilities are a liability

Open Web Application Security Project (OWASP)

Works to improve web security



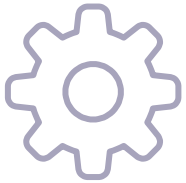
Lots of resources, tools, documents

OWASP Top 10

1. *Broken Access Control*
2. *Cryptographic Failures*
3. *Injection*
4. *Insecure Design*
5. *Security Misconfiguration*
6. *Vulnerable and Outdated Components*
7. *Identification and Authentication Failures*
8. *Software and Data Integrity Failures*
9. *Security Logging and Monitoring Failures*
10. *Server-Side Request Forgery (SSRF)*

2021 OWASP Top 10 https://owasp.org/Top10/

Time to code!



alisaduncan/angular-security-workshop

1. Fork & clone the repo, open in IDE
2. Open 2 terminals for *app* and *`server`*. Navigate into each directory.
3. Run `npm ci` and `npm start` for both
4. Login and inspect what's going on
   a. Username: "admin" or "member"
   b. Password: ???

Recap

- *Beautiful* cookie e-commerce site
- Authentication with username/password
- No access control checks
- Security by obscurity

2.
Safeguard views

Understand attack vectors
Angular XSS prevention model
Build DOM elements
Sanitize DOM bindings

Injection attacks

Unintentionally running malicious code positioned by an attacker, thus allowing unauthorized actions

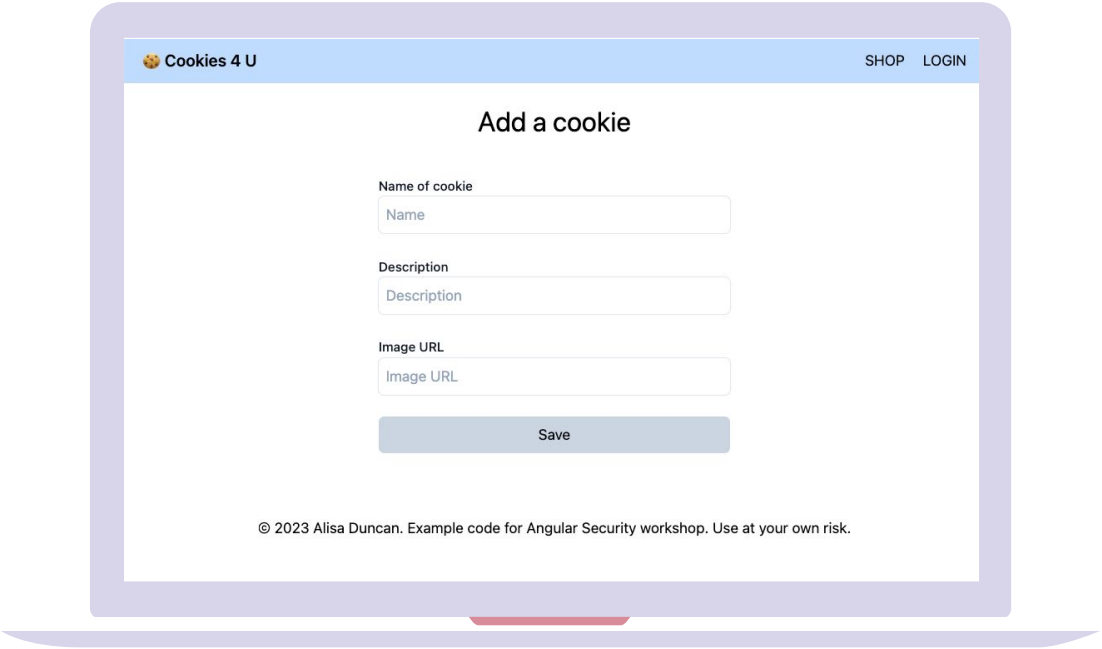
Cross-Site Scripting (XSS)

Occurs when there's not good data and code boundaries in the values we incorporate within web applications

Example attack

Attackers insert their malicious code into your application through legitimate means.

```
<img src=1 onerror="alert('Eek!')" />
```



Incorporate values in web apps

```
<img src=1 onerror="alert('Eek!')" />
```

Angular treats all values as untrusted

Interpolation escapes code

value = `<img src=1 onerror="alert('Eek!')" />`;

```
<p>
  {{value}}
</p>
```

```
<p>
  &lt;img src=1
onerror=&quot;alert('
Eek!')&quot; /&gt;
</p>
```

Property binding sanitizes code

value = `<img src=1 onerror="alert('Eek!')" />`;

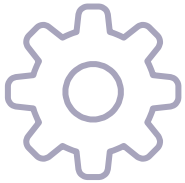
```
<p
 [innerHTML]="value">
</p>
```

```
<p>
  <img src=1 />
</p>
```

Experiment!

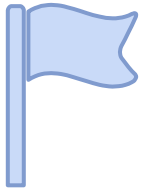
1. Browse the shop in the app.
2. Open *`server/server.js`*.
3. Comment out the imageUrl and uncomment `imageUrl` with `svg` for Chocolate Chip cookie.
4. Stop and restart server (server does not have live reload).
5. Open developer tools console and reload the shop. Experiment!

Time to code!



1. In *`server/server.js`*, delete the Chocolate Chip cookie `imageUrl` with the `svg` and uncomment the starting image.
2. Open *`app/src/app/shop/shop.component.ts`*.
3. Replace property binding for `<img [src]="" />` with interpolation.

Checkpoint



```
<div class="w-36 relative">
  <img
    src="{{product.imageUrl}}"
    alt="{{product.name}}"
  />
</div>
```

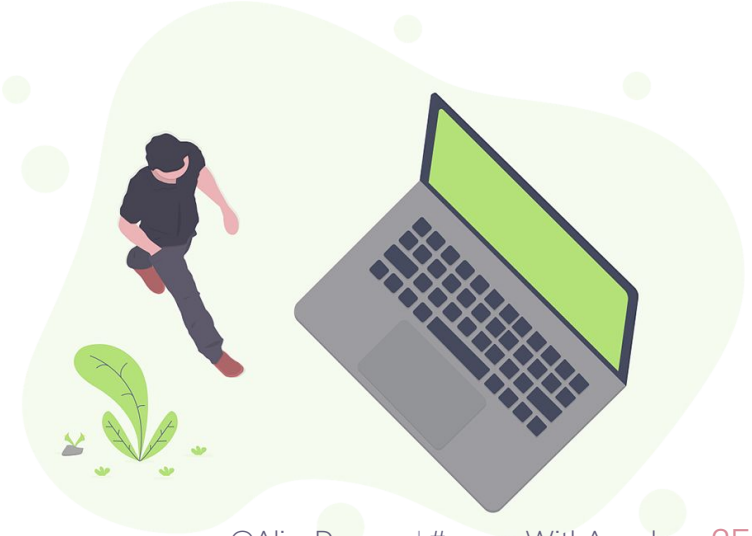
Build DOM elements

Always

Use Angular templates to construct the DOM

Avoid

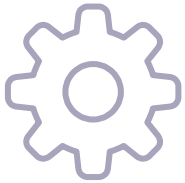
- Building DOM by hand
- Construct DOM using strings
- Use server-side templating engines to create DOM string



Experiment!

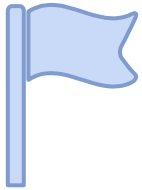
1. Search for a cookie on the homepage of the app, such as "Chocolate chip"
2. Don't wait for search results-it doesn't work. We still see the gist of what should happen.
3. Search for `<img src=1 onerror="alert('Eek!')" />`
4. Open *`results/results.component.ts`*.
5. Do you see the problem? How do you propose fixing it?

Time to code!



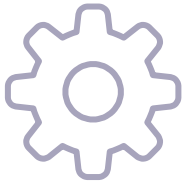
1. Open *`results/results.component.ts`*.
2. Find the `div` with the template variable `#term`.
3. Replace `#term` with interpolation or property binding to `innerHTML` using `query` input parameter.
4. Remove `ViewChild` declaration and `AfterViewInit` lifecycle hook.

Checkpoint



```
@Component({
 selector: 'app-results',
 template: `
  <p>We're on it! Searching our kitchen for...</p>
  <div [innerHtml]="query"></div>
`})
export class ResultsComponent implements OnInit {
  @Input() query = '';
  private productsService = inject(ProductsService);
  ngOnInit(): void { /* implementation here */ }
}
```

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `safeguard-views`

Recap

- Angular has built-in XSS protection
- Always use Angular templates to build the DOM
- Appropriately use interpolation and property binding

3.
Safely input

Understand attack vectors
Sanitize user input
Validate input
Bypass security measures

How does malicious code get in?

Various forms of user input

- Form input
- URL parameters
- External resources
- Attackers are inventive

Correlates to XSS attack forms

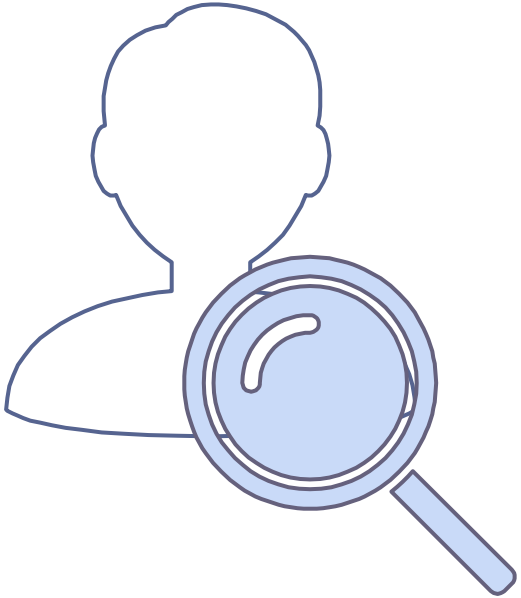
Stored XSS

Reflected XSS

DOM XSS

Treat values as untrusted

Always sanitize and validate user entry



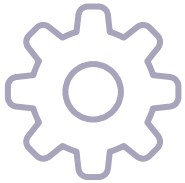
Validate input

- Limit user input to allow lists
  - ★ Form control validators
- Limit input to minimum requirement for use
  - ★ Construct values by minimally relying on input

Experiment!

1. View app in browser & open dev tools. Navigate to "/admin" by typing in route in browser location.
2. Add a new cookie using any value. View cookie in "/shop".
3. Is this a safe operation? Can we trust the user input? In what ways are we insecure?

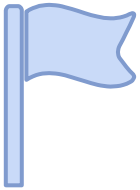
Time to code!



1. Open *`admin/admin.component.ts`*.
2. Add validators to `imageUrl` form control. Add the following:
   a. `Validators.required`
   b. `Validators.pattern(`

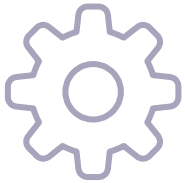
```
/https:\/\/[a-z0-9\d.\/\-?=&]*/mi
)
```

Checkpoint



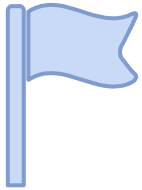
```
@Component({...})
export class AdminComponent {
  // remaining properties
  public imageUrl = new FormControl('', [
    Validators.required,
    Validators.pattern(/https:\/\/[a-z0-9\d.\/\-?=&]*/mi)
  ]);
}
```

Time to code!



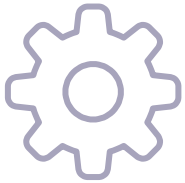
1. Create a new file for an `imageUrl` validator: *`/admin/validator.ts`*
2. Declare a validator function `imageUrlValidator` taking a domain as a param :
   ```
   export function imageUrlValidator(domain: string): ValidatorFn { }
   ```
3. The validator returns a function with an `AbstractControl` param returning `ValidationErrors | null`:
   ```
   return (control: AbstractControl): ValidationErrors | null => { }
   ```
4. The function checks the control for the domain and returns an error or null:
   ```
   const includesDomain = (control.value as string).includes(domain);
   return includesDomain ? null : { imageUrl: { value: control.value
   }}
   ```

Checkpoint



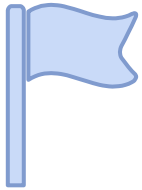
```
export function imageUrlValidator(domain: string): ValidatorFn {
  return (control: AbstractControl) : ValidationErrors | null => {
    const includesDomain =
      (control.value as string).includes(domain);
    return includesDomain ?
      null : {imageUrl: {value: control.value}};
  }
}
```

Time to code!



1. Open *`admin/admin.component.ts`*.
2. Add new validator to `imgUrl` form control. Add the following:
   a. `imageUrlValidator('unsplash.com')`

Checkpoint



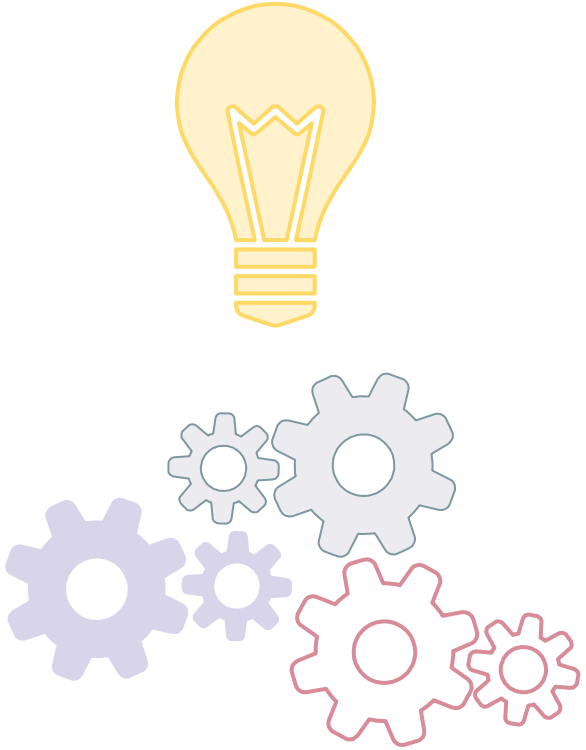
```
@Component({...})
export class AdminComponent {
  // existing properties
  public imageUrl = new FormControl('', [
    Validators.required,
    Validators.pattern(/https:\/\/[a-z0-9\d.\/\-?=&]*/mi),
    imageUrlValidator('unsplash.com')
  ]);
}
```

Experiment!

1. Try adding a cookie in the form. What URL patterns work? You will not see an error display because there is no handling to display errors.

Consider the APIs

The app should help prevent attacks on the API backend where possible



Experiment!

1. View app in browser. Open developer tools. Search for a product.
2. Notice the network call includes a product search.
3. Open *`app/src/results/results.component.ts`* to see what's going on.
4. Search for "gingersnap; DROP TABLE Users;"
5. What can we do to prevent this SQL injection attack?

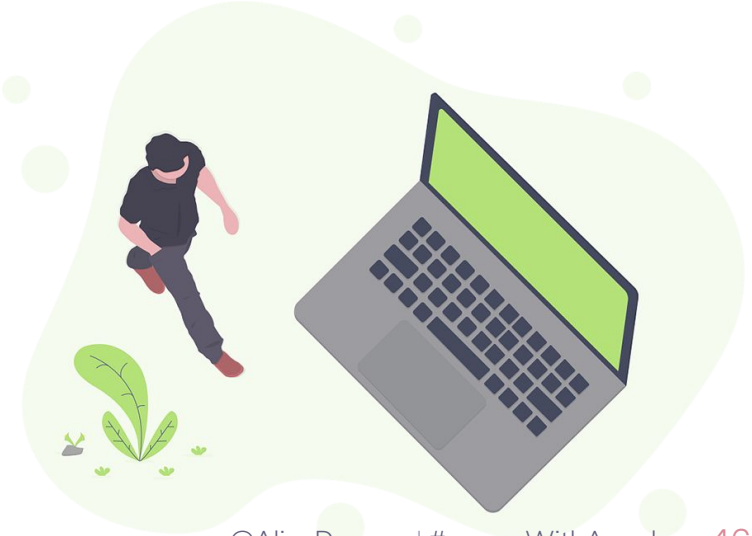
Use the DOM sanitizer

Explicitly sanitize

Sanitize values before adding to templates

Bypass sanitization

Mark values as trusted and bypass sanitization explicitly to skip built-in sanitization



Angular's `DOMSanitizer`

```
@Component({
  selector: 'app-results',
  template: `<div #term></div>`
})
export class ResultsComponent implements AfterViewInit {
  @ViewChild('term') public el!: ElementRef<HTMLElement>;
  @Input() query = '';
  private sanitizer = inject(DomSanitizer);

  ngAfterViewInit(): void {
    const s = this.sanitizer.sanitize(SecurityContext.HTML, query);
    this.el.nativeElement.innerHTML = s;
  }
}
```

Security context

NONE

HTML

STYLE

SCRIPT

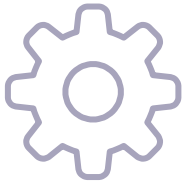
URL

RESOURCE URL

Experiment!

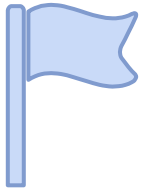
1. Open *app/src/home/banner.component.ts*. Notice how the banner appends to the DOM.
2. Open *app/src/promos.service.ts* and find `getBanner()`. Remove the the `_target="blank"`. Change the `<a href>` value from google.com to "`javascript:alert('Oh no!')`".
3. Promos come from marketing. Should we trust this input?
4. Replace the link to google.com

Time to code!



1. Open *`app/src/home/banner.component.ts`*.
2. Inject `DomSanitizer`.
3. In `ngAfterViewInit()`, sanitize the message using the `DomSanitizer` and `SecurityContext.HTML`.
4. Add the sanitized message to the `ElementRef`'s `innerHTML`.

Checkpoint



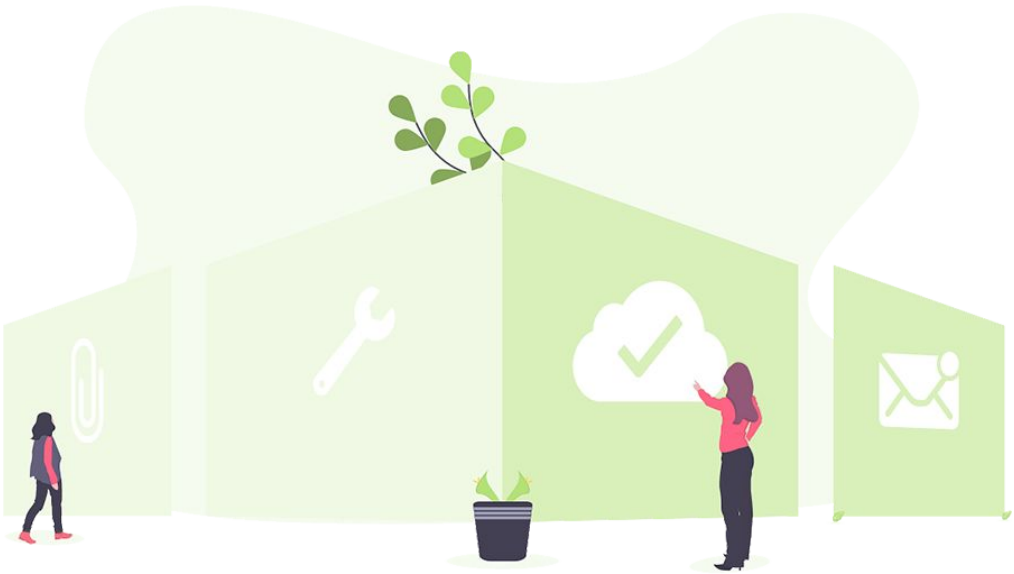
```
export class BannerComponent implements AfterViewInit {
  @ViewChild('banner') public el!: ElementRef<HTMLElement>;
  private promosService = inject(PromosService);
  private sanitizer = inject(DomSanitizer);

  ngAfterViewInit(): void {
    const message = this.promosService.getBanner();
    const sanitized = this.sanitizer.sanitize(
                        SecurityContext.HTML, message
                      );
    this.el.nativeElement.innerHTML = sanitized ?? '';
  }
}
```

Experiment!

1. Run the application.
2. Does the banner look different? What changed?

Marketing wants a new feature

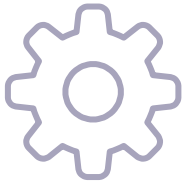
We want to promote the coffee shop by showing different videos



Experiment!

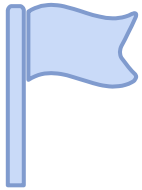
1. Open *`app/src/home/coffee-promo.component.ts`*. Notice the YouTube video is static.
2. Marketing wants to send the video link as part of the coffee promo. Move the video link to a property and try property binding it. What happens? What do you see in developer tools console?
3. Open *`promos.service.ts`*. Notice promos include a YouTube video id.

Time to code!



1. Open *`home/coffee-promo.component.ts`*.
2. Create a variable for `videoLink!: SafeResourceUrl` and property bind `iframe`'s `src` attribute.
3. Inject `DomSanitizer`.
4. In `ngOnInit()`, use object deconstruction to get `videoId` within the `{message} variable`.
5. Set `videoLink` variable with `DomSanitizer`'s method: `sanitizer.bypassSecurityTrustResourceUrl(`https://www.youtube.com/embed/${videoId}`);`.

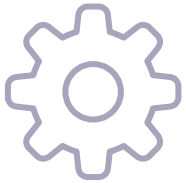
Checkpoint



```
@Component({ selector: 'app-coffee-promo',
  template: `...<iframe [src]="videoLink"...></iframe>...`
})
export class CoffeePromoComponent implements OnInit {
  videoLink!: SafeResourceUrl;
  private sanitizer = inject(DomSanitizer);
  private promosService = inject(PromosService);

  public ngOnInit(): void {
    const {message, videoId} = this.promosService.getCoffeePromos();
    // existing implementation
    this.videoLink = this.sanitizer.bypassSecurityTrustResourceUrl(
      `https://www.youtube.com/embed/${videoId}`
    );
  }
}
```

Get a security audit when bypassing sanitization

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `safely-input`

Recap

- Always validate and sanitize values
- Limit attack vectors by limiting input
- Prefer Angular built-in sanitization, and fallback to `DOMSanitizer`

Discussion

How do you protect against XSS? Have you had to manually build a DOM?

4. Elevate authentication

Understand attack vectors
Delegate to a reputable Identity Provider
Intro to OAuth 2.0 + OIDC
Elevating authentication factors

Use a reputable Identity Provider

Identity Providers help you manage authentication, authorization, and user identities **securely**

Use industry standard protocols

OAuth 2.0

OAuth 2.0 is a industry best practice for authorization.

OpenID Connect

OpenID Connect (OIDC) is an identity layer on top of OAuth 2.0 to support authentication and identity.

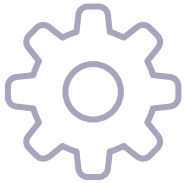
Elevate authentication factors

Passwordless auth using FIDO2 and WebAuthn or Passkeys offers the most secure authentication mechanism.

Secure authentication

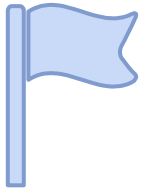
1. Create a free Okta developer account
2. Add OpenID Connect (OIDC) certified library
3. Try passwordless (optional)

Time to code!



1. Create a free Okta developer account at developer.okta.com.
2. In Okta dashboard sidebar go to **Applications** > **Applications**.
3. Press **Create App Integration**. Select **OIDC** and **Single-Page App**.
4. Change your **Sign-in redirect URIs** to http://localhost:4200 and **Sign-out redirect URIs** to http://localhost:4200
5. In **Assignments**, select **Allow everyone in your org to access**.
6. In sidebar, go to **Security** > **API**, select **Trusted Origins**. Press **+ Add origin**. Enter "My SPA" for **name**, http://localhost:4200 for **URL**, and select **CORS** and **Redirect** types.

Checkpoint



APPLICATION

App integration name	My SPA
Application type	Single Page App (SPA)
Grant type	Client acting on behalf of a user
	☑ Authorization Code
	☑ Refresh Token
	☐ Implicit (hybrid)

LOGIN

Sign-in redirect URIs	☐ Allow wildcard * in login URI redirect.
	http://localhost:4200
Sign-out redirect URIs	http://localhost:4200

API

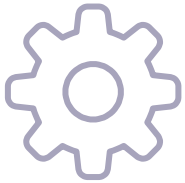
Authorization Servers | Tokens | Trusted Origins

Filters	Name	Origin URL	Type
All	My SPA	http://localhost:4200	CORS Redirect
CORS			
Redirect			
iFrame embed			

Okta supports OAuth 2.0 + OIDC

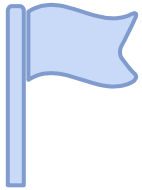
We can use a OIDC-certified library to replace the app's login and delegate authentication factors to Okta.

Time to code!



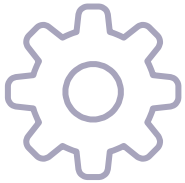
1. Add OIDC-certified library using the schematics in the terminal for the app: `ng add angular-auth-oidc-client`
   a. Select OIDC Code Flow PKCE using refresh tokens
   b. Enter authority: https://{yourOktaDomain}/oauth2/default
2. Configure the *`app/src/app/auth/auth-config.module.ts`* using values from Okta application:
   a. Set the `clientId` value with your Okta application Client ID
   b. Set scope to the example: `'openid profile offline_access'`

Checkpoint



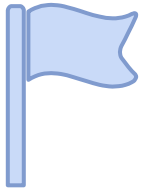
```
@NgModule({
  imports: [AuthModule.forRoot({
    config: {
      authority: 'https://{yourOktaDomain}/oauth2/default',
      …
      clientId: '{yourOktaClientID}',
      scope: 'openid profile offline_access',
      …
    }
})],
exports: [AuthModule],
})
```

Time to code!



1. Set up the required application-wide authentication check Observable in *`app.component.ts`* by:
   a. Inject the `OidcSecurityService`.
   b. Add the following code to the constructor:
   `this.oidcService.checkAuth().pipe(takeUntilDestroyed())`
   `.subscribe(res => console.log(res));`
   c. Import `takeUntilDestroyed` from `@angular/core/rxjs-interop` if needed.

Checkpoint

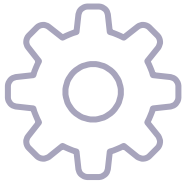


```
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';

export class AppComponent {
  private oidcService = inject(OidcSecurityService);

  constructor() {
    this.oidcService.checkAuth().pipe(
      takeUntilDestroyed()
    ).subscribe( res => console.log(res));
  }
}
```

Delegate user consent

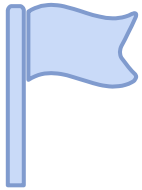
Redirect to the Identity Provider to authenticate. Free yourself from password storage headaches and implementing non-phishable authentication factors.

Time to code!



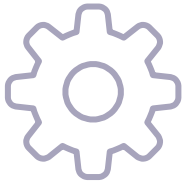
1. Open *`auth.service.ts`* to add the OIDC library change to the existing methods. Delete the `URL`, `http`, and `userInfo` properties.
2. Inject the `OidcSecurityService` in the service.
3. Change `login()` by deleting credentials parameters, set return type to void, and to use the `OidcService`: `this.oidcService.authorize();`
4. Change the `logout()` by setting return type to `void` and using service: `this.oidcService.logoff().pipe(take(1)).subscribe();`
5. Change `isAuthenticated()` to a `readonly` property to use the `OidcService`'s `isAuthenticated$` observable property: `this.oidcService.isAuthenticated$.pipe(map(res => res.isAuthenticated));`

Checkpoint



```
export class AuthService {
  private oidcService = inject(OidcSecurityService);

  readonly isAuthenticated = this.oidcService.isAuthenticated$
    .pipe(map(res => res.isAuthenticated));

  login(): void {
    this.oidcService.authorize();
  }

  logout(): void {
    this.oidcService.logoff().pipe(take(1)).subscribe();
  }
}
```

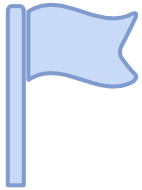
Time to code!



1. Open *`header/header.component.ts`* to use `AuthService` changes. You will have IDE errors after this slide.
2. Add an `isAuthenticated$` property: `isAuthenticated$ = this.authService.isAuthenticated.pipe(takeUntilDestroyed());`.
3. Add an `onLogin(): void` method calling: `this.authService.login();`.
4. Update `onLogout()` to call the updated method: `this.authService.logout();`.
5. Update login code in template using the new isAuthenticated$ property:

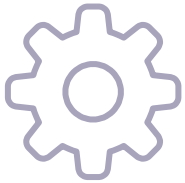
```
<li *ngIf="!(isAuthenticated$ | async); else logout">
    <a (click)="onLogin()" class="uppercase">Login</a>
</li>
```

Checkpoint



```
@Component({...
  template: `...<li *ngIf="!(isAuthenticated$ | async); ...">
    <a (click)="onLogin()"> ...`
})
export class HeaderComponent {
  private authService = inject(AuthService);
  isAuthenticated$ =
    this.authService.isAuthenticated.pipe(takeUntilDestroyed());

  onLogin(): void { this.authService.login(); }

  onLogout(): void { this.authService.logout(); }
}
```

Time to code!

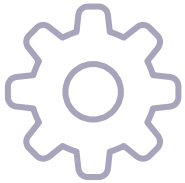


1. Delete the *`login/login.component.ts`* component and directory.
2. Remove the /login path from routing and delete imports.
3. Remove `LoginComponent` from `AppModule` declarations and delete imports.

Experiment!

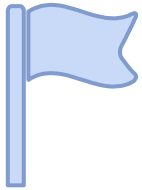
1. Verify logging in takes you to Okta and redirects you back to the application.
2. Ensure the logout link shows up in the navigation bar.

Time to code!



1. Open *`server/server.js`* and delete the following:
   a. User list
   b. Anything using `passportLocal`, `passport`, or `express-session`, including the `require` statements
   c. The `'/api/signin'` and `'/api/signout'` routes
   d. The `checkAuthenticated` method and `'/api/users'` route

Checkpoint



```
const express = require('express')
const cookieParser = require('cookie-parser');
const cors = require('cors');

/* app, port, and products declaration */

app
.use(cors())
.use(cookieParser())
.use(express.json())
.listen(port, () => { /* code */ });

app.get('/api/xsrfEndpoint', (req, res, next) => { /* code */ }
...
```

80%

web application breaches stem from stolen credentials

2022 Verizon Data Breach Investigations verizon.com/business/resources/reports/dbir/

>100

The number of passwords managed by the average person

NordPass study https://nordpass.com/

61%

Increase in phishing attacks in 2022

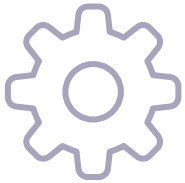
2022 Phishing Landscape study https://interisle.net/PhishingLandscape2022.html

What if we stopped using passwords?

Experiment!

1. Try passwordless in an Okta account. Change the `authority` and `clientId` in the `AuthModule` to use:
   ```
   authority: https://dev-32413740.okta.com/oauth2/default
   clientId: 0oadc4pzvwDj9W6Vv5d7
   ```
2. Try signing in. You have an option to sign in with biometrics.
3. Revert the `authority` and `clientId` back for your account.

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `elevate-authentication`

Recap

- Use industry standard protocols and reputable Identity Providers
- Don't write your own auth code!
- Elevate authentication mechanisms by moving away from passwords

5.
Secure API calls

Understand attack vectors
Add Authorization header to API calls
Validate access token

> ...94% of applications were tested for some form of broken access control...

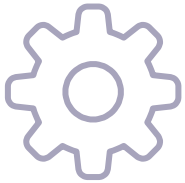
A02:2021 - Broken Access Control
https://owasp.org/Top10/A01_2021-Broken_Access_Control/

Access tokens authorize requests for data and actions

Experiment!

1. Open *`server/server.js`*. Should anyone be able to add new products to the application?

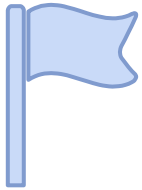
Time to code!



1. Create an interceptor using Angular CLI: `ng g interceptor auth`
2. Open *`app/auth.interceptor.ts`*
3. Inject the `OidcSecurityService`.
4. Get access token from `OidcSecurityService` and set it in Authorization header within the `intercept()` before the return statement:

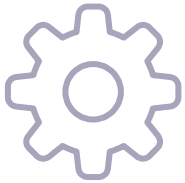
```
this.oidcService.getAccessToken()
.pipe(take(1))
.subscribe( t => {
  const headers = request.headers.set('Authorization',
`Bearer ${t}`);
  request = request.clone({headers});
});
```

Checkpoint



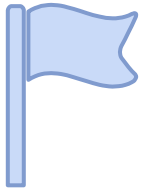
```
export class AuthInterceptor implements HttpInterceptor {
  private oidcService = inject(OidcSecurityService);
  intercept(request, next): Observable<HttpEvent<unknown>>{
    this.oidcService.getAccessToken().pipe(take(1))
    .subscribe( t => {
      const headers =
      request.headers.set('Authorization', `Bearer ${t}`);
      request = request.clone({headers})
    });
    return next.handle(request);
  }
}
```

Time to code!



1. Open *`app.module.ts`* so you can register the interceptor.
2. Add a new provider to the providers array:
   `{ provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }`
3. Ensure you import your `AuthInteceptor` implementation, not the one from the OIDC client library.

Checkpoint



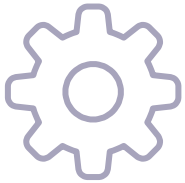
```
import { AuthInterceptor } from './auth.interceptor';

@NgModule({
  declarations: [], imports: [], bootstrap: [AppComponent],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true
    }
  ],
})
```

Validate authorized calls by verifying access tokens

Check validity of token

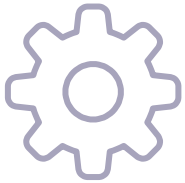
Verify access

Time to code!



1. Add JWT verifier library from Okta to server project:
   `npm i @okta/jwt-verifier`
2. Open *server/server.js*.
3. Include library:
   ```
   const OktaJwtVerifier = require('@okta/jwt-verifier');
   ```
4. Create a new instance of the verifier and pass in the issuer:
   ```
   const oktaJwtVerifier = new OktaJwtVerifier({
     issuer: 'https://{yourOktaDomain}/oauth2/default'
   });
   ```

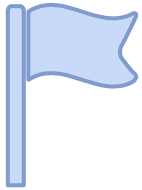
Time to code!



1. Add method to ensure the request includes Authorization header :

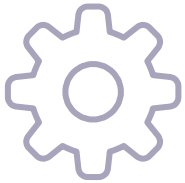
```
const checkAuthorized = async (req, res, next) => {
  const authHeader = req.headers.authorization || '';
  const match = authHeader.match(/Bearer (.+)/);
  if (!match) { return res.status(401).send(); }
  const accessToken = match[1];
  if (!accessToken) { return res.status(401).send();}
  next();
}
```

Checkpoint



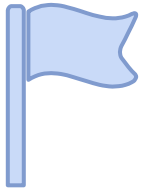
```
const OktaJwtVerifier = require('@okta/jwt-verifier');

const oktaJwtVerifier = new OktaJwtVerifier({
  issuer: 'https://{yourOktaDomain}/oauth2/default'
});

const checkAuthorized = async (req, res, next) => {
  const authHeader = req.headers.authorization || '';
  const match = authHeader.match(/Bearer (.+)/);
  if (!match) { return res.status(401).send(); }

  const accessToken = match[1];
  if (!accessToken) { return res.status(401).send(); }

  next();
}
```

Time to code!



1. In *server.js*, verify access token in the `checkAuthorized` method after getting the access token from the Authorization header:
```
try {
  await oktaJwtVerifier.verifyAccessToken(accessToken,
'api://default');
} catch (err) { return res.status(403).send(err.message); }
```
2. Add check to products routes that require authorization:
```
app.route('/api/products').get((_, res) => res.json(products))
.post(checkAuthorized, (req, res) => { /* existing code here
*/ });
```

Checkpoint

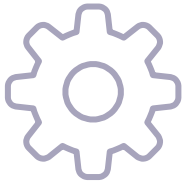


```
const checkAuthorized = async (req, res, next) => {
  // prior implementation to get access token from request
  try {
    await oktaJwtVerifier.verifyAccessToken(accessToken, 'api://default');
  } catch(err) { return res.status(403).send(err.message); }

  next();
}

app.route('/api/products')
.get((_, res) => res.json(products))
.post(checkAuthorized, (req, res) => { /* POST implementation */ });
```

Protect access tokens

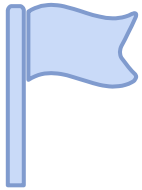
Send access tokens only to allowed APIs. This is especially important if you call multiple APIs.

Time to code!



1. In *`auth.interceptor.ts`*, create a list of allowed origins within the intercept():
   ```
   const allowedOrigins = ['localhost:3000'];
   ```
2. Wrap the existing code to add the access token to Authorization header with a check to ensure the requested URL is in the allowed origins list:
   ```
   if (!!allowedOrigins.find(origin =>
     request.url.includes(origin))) {
       // prior implementation here
   }
   return next.handle(request);
   ```

Checkpoint

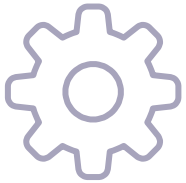


```
intercept(request, next): Observable<HttpEvent<unknown>> {
  const allowedOrigins = ['localhost:3000'];

  if (!!allowedOrigins.find(origin => request.url.includes(origin))){
    // prior implementation here
  }

  return next.handle(request);
}
```

Experiment!

1. Restart the server and test application. Open developer tools to the Network tab.
2. Try signing in and navigate to the "/admin" route to add a cookie.
3. Do you see the access token in the Authorization header?

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `secure-api`

Recap

- Verify access to resources
- Authorization context depends on your access control model
- Prevent access token leaks by limiting the allowed origins

Discussion

Do you utilize OAuth and OIDC in your application? What kind of access checks do you have in your APIs?

6.
Defend cookies

Understand attack vectors
Consider sessions
Prevent XSRF attacks

Watch out for

Cross-Site Request Forgery (CSRF)

Image by Karsten Winegeart on unsplash.com

Example CSRF attack

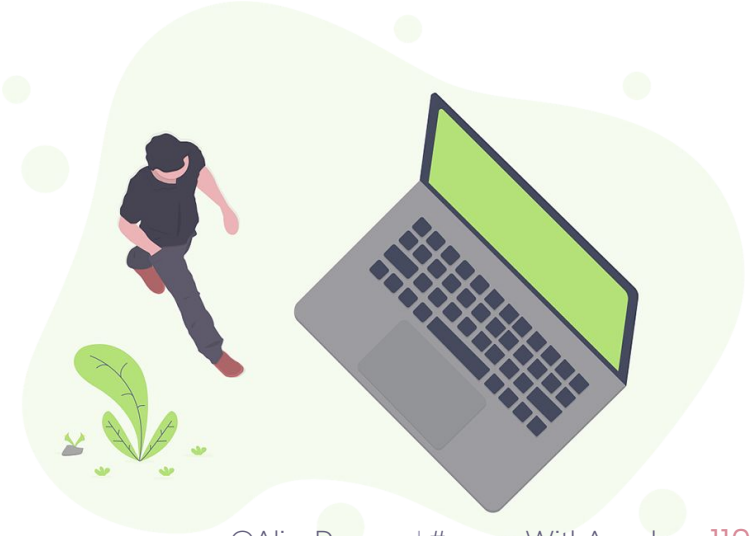
Mitigate CSRF attacks

Browser protection

Utilize the `sameSite` and `HttpOnly` properties on cookies, depending on use case. Ensure the browsers your app supports also supports these cookie properties.

Mitigation strategy

Send a CSRF token in addition to session cookie that backend systems validate prior to processing requests. This pattern is known as the Double-Submit Cookie pattern.



Angular's CSRF protection

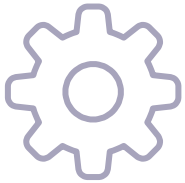
```
@NgModule({
  declarations: [...],
  imports: [...,
   HttpClientXsrfModule.withOptions({
      cookieName: 'XSRF-TOKEN',
      headerName: 'X-XSRF-TOKEN'
  })],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

```
bootstrapApplication(AppComponent, {
  providers: [...,
  provideHttpClient(
    withXsrfConfiguration({
      cookieName: 'XSRF-TOKEN',
      headerName: 'X-XSRF-TOKEN'
    })
  )]
})
.catch(err => console.error(err));
```

Experiment!

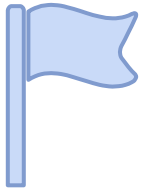
1. Open *`server/server.js`*. Find the `/api/xsrfEndpoint` route. Notice the cookie has the `sameSite` property set to strict. Calls to the API must run on the same site for this cookie.
2. We are only demonstrating how Angular sends the `X-XSRF-TOKEN` header. Use implementation from this section in your production apps at your own risk!

Time to code!

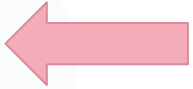


1. In `server/server.js`, find the Express app middleware to use `cors()` and remove it. We won't need to enable CORS going forward.
2. Find the POST products endpoint to add a comment about checking the XSRF token. We are not implementing the check, but production apps should add XSRF handling.

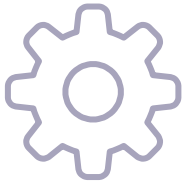
Checkpoint



```
app
.use(cookieParser())
.use(express.json())
.listen(port, () => {
  console.log(`Listening on port ${port}`);
});

app.route('/api/products')
.get((_, res) => res.json(products))
.post(checkAuthorized, (req, res) => {
  // don't forget to check XSRF
  /* remaining implementation */
});
```

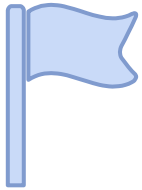


Time to code!



1. Open *`app/src/app.module.ts`*, and add a function for the `APP_INITIALIZER` injection token to run before the app declaration:
```
function xsrfTokenFactory(http: HttpClient): () =>
Observable<any> {
  return () => http.get('localhost:3000/api/xsrfEndpoint');
}
```
2. Import the `HttpClientXsrfModule` to automatically add the XSRF token handling.
3. Provide the `APP_INITIALIZER` to retrieve the XSRF token cookie
```
{ provide: APP_INITIALIZER, useFactory: xsrfTokenFactory,
deps: [HttpClient], multi: true }
```

Checkpoint



```
function xsrfTokenFactory(http: HttpClient): () => Observable<any> {
  return () => http.get('localhost:3000/api/xsrfEndpoint');
}

@NgModule({
  declarations: [...], bootstrap: [AppComponent],
  imports: [...,
    HttpClientXsrfModule
  ],
  providers: [...,
  { provide: APP_INITIALIZER, useFactory: xsrfTokenFactory, deps:
[HttpClient], multi: true },
  ]
}) export class AppModule { }
```

Experiment!

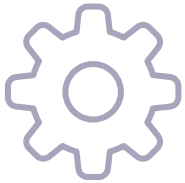
1. Restart the server and run the application. Open developer tools and look at the Network calls. What do you see?

The Angular app and API must run on the same port

Runs on port 4200

Runs on port 3000

Time to code!



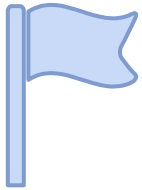
1. Create a file for proxy configuration in *app/src/proxy.conf.json*
2. Add the following properties:

```
{
    "/api": {
        "target": "http://localhost:3000",
        "secure": false
    }
}
```

3. Open app/angular.json and add the following properties in serve:

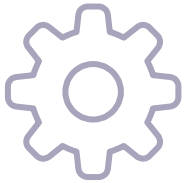
```
"options": {
    "proxyConfig": "src/proxy.conf.json"
}
```

Checkpoint



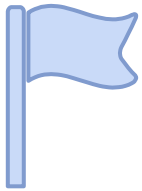
```
"serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
        "proxyConfig": "src/proxy.conf.json",
    },
    "configurations": {...},
    "defaultConfiguration": "development"
}, ...
```

Time to code!



1. Remove the `'localhost:3000'` text from the API URL in the following locations (the URL format should look like `'/api/<endpoint>'`):
   a. *`app.module.ts`* - `xsrfTokenFactory` method
   b. *`products.service.ts`* - URL variable

Checkpoint



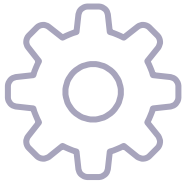
```
function xsrfTokenFactory(http: HttpClient): () =>
Observable<any> { return () => http.get('/api/xsrfEndpoint'); }
```

```
private readonly URL = '/api';
```

Experiment!

1. Stop serving the Angular application and restart it.
2. In the browser, open developer tools to inspect network requests.
3. Add a cookie product in the "/admin" section of the app and watch the network call
4. Do you see the XSRF token in the HTTP header?

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `defend-cookies`

Recap

- Angular automatically adds CSRF protection in outgoing requests
- API server must verify the CSRF token
- Proxy calls in Angular during local development

7.
Guard routes

Understand attack vectors
Examine and configure identity claims
Protect unauthorized access

> Access control enforces policy such that users cannot act outside of their intended permissions.

A02:2021 - Broken Access Control
https://owasp.org/Top10/A01_2021-Broken_Access_Control/

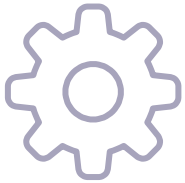
Guard sensitive routes

Protect routes based on the access control measure appropriate for your software system.

Experiment!

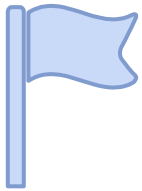
1. Notice you can navigate to "/members" route to view the contents of the Cookie Jar without authenticating. Shouldn't we guard this route to only authenticated users?
2. Notice you can manually navigate to "/admin" route without authenticating. Sounds suspicious...

Time to code!



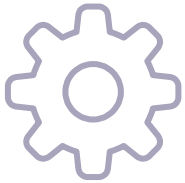
1. Create a file for route guards: *`app/src/app/guards.ts`*
2. Define and export a guard to validate authenticated state in the file:
   `export const authenticatedGuard: CanActivateFn = (route, state) => true;`
3. Inject the `AuthService` as a parameter:
   `(route, state, authService = inject(AuthService)) => true;`
4. Return the `AuthService`'s `isAuthenticated` property:
   `(params) => authService.isAuthenticated;`

Checkpoint



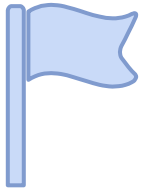
```
export const authenticatedGuard: CanActivateFn =
  (route, state, authService = inject(AuthService)) =>
  authService.isAuthenticated;
```

Time to code!



1. Open *`app-routing.module.ts`* to incorporate the guards in the routes.
2. To the `'members'` path, add the `canActivate` property and `authenticatedGuard` in the route definition:
   `{ path: 'members', component: MembersComponent, canActivate: [authenticatedGuard] }`
3. Add the `canActivate` property with `authenticatedGuard` to the `'admin'` path definition.

Checkpoint



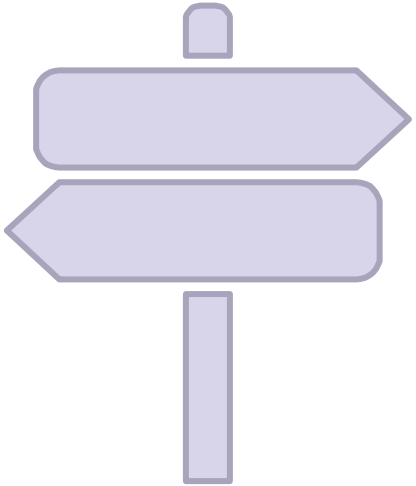
```
const routes: Routes = [..., {
  path: 'members', component: MembersComponent,
  canActivate: [authenticatedGuard]
},{
  path: 'admin', loadChildren: /* load module */,
  canActivate: [authenticatedGuard]
}];
```

Experiment!

1. Verify the guards prevent unauthenticated users from navigating to those routes.

Use access controls measures

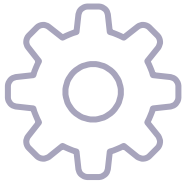
Access control measures will be different for each software system. ID token claims can be a measure for conditional access.



Experiment!

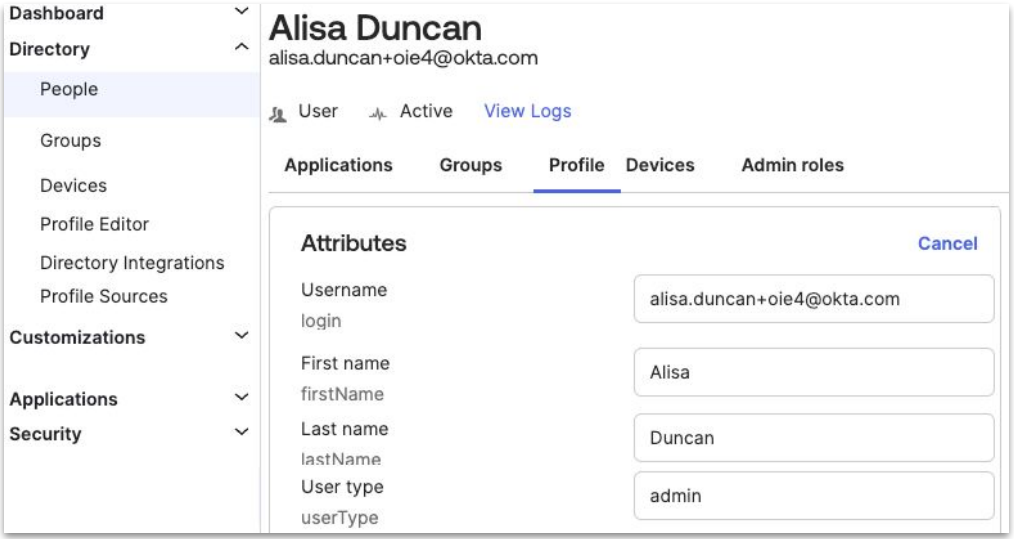
1. All authenticated users can manually navigate to the "/admin" route. Should they? Or should access control measures change?

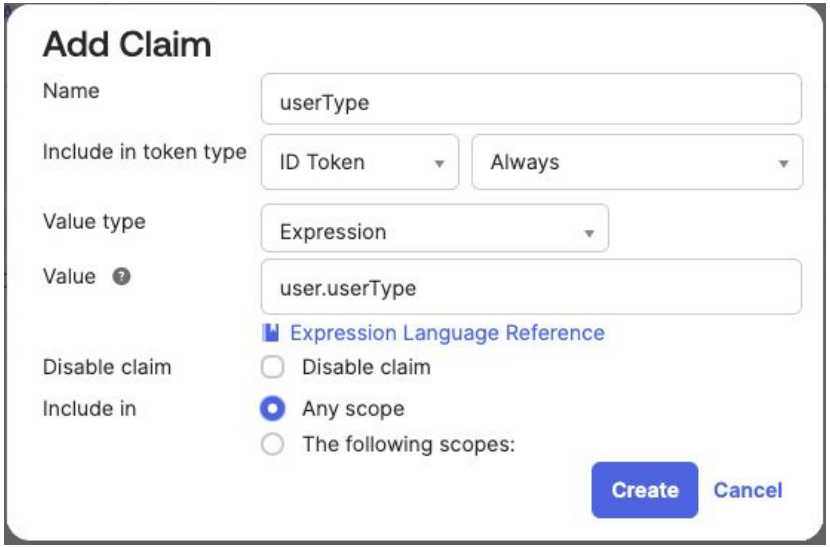
Time to code!



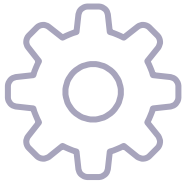
1. Add a value to the userType claim in Okta for conditional access. In the Okta sidenav, navigate to **Directory > People** and select your user to edit.
2. In your user profile, select **Profile** tab and press **Edit**. Scroll down to **User type** and enter "admin" as the value.
3. Navigate to **Security > API** in Okta. On the **Authorization Servers** tab, find the entry named "default" and press the **pencil icon** to edit.
4. In the default settings, view the **Claims** tab and press the **+ Add Claim** button.
5. In the **Add Claim** dialog, use "userType" for Name, select **ID Token** in **Include in token type**, and set the **Value** to `user.userType`. Press **Create**.

Checkpoint


Dashboard
Directory
People
Groups
Devices
Profile Editor
Directory Integrations
Profile Sources
Customizations
Applications
Security
Alisa Duncan
alisa.duncan+oie4@okta.com
User
Active
View Logs
Applications
Groups
Profile
Devices
Admin roles
Attributes
Cancel
Username
login
alisa.duncan+oie4@okta.com
First name
firstName
Alisa
Last name
lastName
Duncan
User type
userType
admin


Add Claim
Name
userType
Include in token type
ID Token
Always
Value type
Expression
Value
user.userType
Expression Language Reference
Disable claim
Disable claim
Include in
Any scope
The following scopes:
Create
Cancel

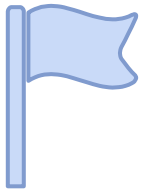
Time to code!



1. Open *app/src/app/auth.service.ts*.
2. Add a new property called `isAdmin`.
3. The `isAdmin` property uses the `OidcSecurityService`'s `userData$` stream to determine if the user is an admin:

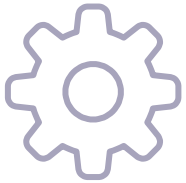
```
readonly isAdmin =
this.oidcSecurityService.userData$.pipe(
  filter(data => !!data && data.userData),
  map(data => data.userData['userType'] === 'admin')
);
```

Checkpoint



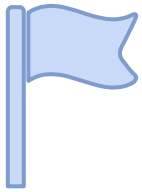
```
readonly isAdmin =
this.oidcSecurityService.userData$.pipe(
  filter(data => !!data && data.userData),
  map(data => data.userData['userType'] === 'admin')
);
```

Time to code!



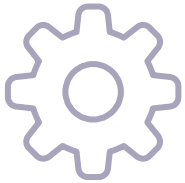
1. Open *`app/src/app/guards.ts`*
2. Define and export a guard to validate if user is an admin:
   `export const adminGuard: CanActivateFn = (route, state) => true;`
3. Inject the `AuthService` as a parameter:
   `(route, state, authService = inject(AuthService)) => true;`
4. Return the `AuthService`'s `isAdmin` property:
   `(params) => authService.isAdmin;`

Checkpoint



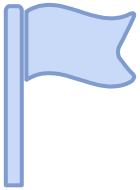
```
export const adminGuard: CanActivateFn =
(route, state, authService = inject(AuthService)) =>
authService.isAdmin;
```

Time to code!



1. Open *`app-routing.module.ts`* to incorporate the `isAdmin` guard in the route.
2. Add the `adminGuard` to the existing `canActivate` guard array in `'admin'` path definition.

Checkpoint

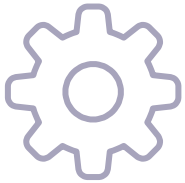


```
const routes: Routes = [..., {
  path: 'admin', loadChildren: /* load module */,
  canActivate: [
    authenticatedGuard,
    adminGuard
  ]
}];
```

Avoid security by obscurity

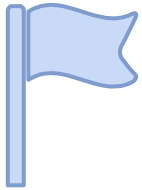
No need to hide the /admin route for "security" purposes. It's time to improve user experience.

Time to code!



1. Open *`header/header.component.ts`*.
2. Define a wrapper property for `isAdmin$` like you have for `isAuthenticated$`:
   ```
   isAdmin$ =
   this.authService.isAdmin.pipe(takeUntilDestroyed());
   ```
3. Add a list item to the nav list for the "/admin" route in the `#logout` template variable if the user is an admin after the "/members" route:
   ```
   <li *ngIf="isAdmin$ | async" class="pr-5">
     <a routerLink="/admin" class="uppercase">Manage</a>
   </li>
   ```

Checkpoint

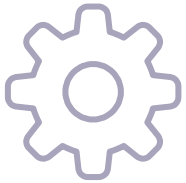


```
@Component({ selector: 'app-header',
  template: `<ng-template #logout> ...
    <li *ngIf="isAdmin$ | async" class="pr-5">
      <a routerLink="/admin" class="uppercase">Manage</a></li>
...</ng-template>`
})
export class HeaderComponent {
  /* existing properties and methods not shown */
  public isAdmin$ = this.authService.isAdmin.pipe(
    takeUntilDestroyed());
}
```

Experiment!

1. Log in and verify the admin route displays and is navigable for you.
2. Log out. Edit the Okta user profile to remove `userType` of "admin". For example, change it to "test".
3. Log in and verify the admin route no longer displays and is not navigable.

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `guard-routes`

Recap

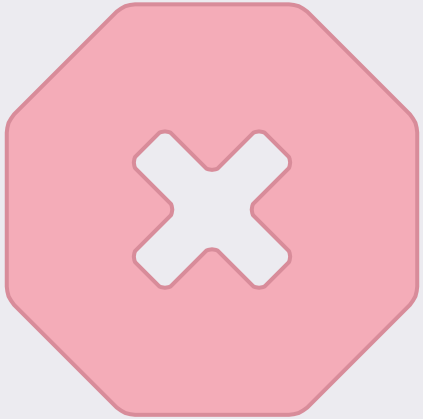
- Adding access controls and guarding assets is the number one OWASP vulnerability.
- Utilize the access control measures appropriate for your system and protect resources, including API calls, appropriately.

Discussion

What access control measures do you use in the application?

8.
Protect views

Understand attack vectors
Secure conditional views



Deny by default

Protect views

Route guards

Protect features and routes based on access control measures.

Built-in structural directives

Use built-in structural directives (or built-in control flow structures) such as `*ngIf` or `@if` for simple use cases.

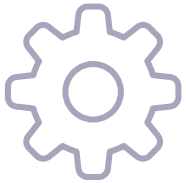
Custom structural directive

When access control needs are more complex, write your own structural directive or utilize switch control flow.

Experiment!

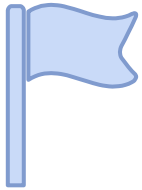
1. The coupon codes that show up on the home page should only display for specific users attributes.
2. Check out *`app/promos.service.ts`* and notice deals have a deal that ties to user attributes.
3. How would you implement displaying the correct deal for the user type?

Time to code!



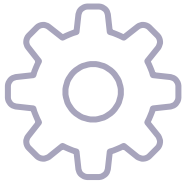
1. Use Angular CLI to create a new directive: `ng g directive user-type`
2. Open *`app/src/app/user-type.directive.ts`*.
3. Add two string private properties named `userType` and `userTypeClaim`.
4. Add an input method named `appUserType` to set the `userType` property:
   ```
   @Input() set appUserType(userType: string) {
   this.userType = userType; }
   ```
5. Add an input method to set the `userTypeClaim` property:
   ```
   @Input() set appUserTypeClaim(userTypeClaim: string) {
   this.userTypeClaim = userTypeClaim; }
   ```
6. Add a property for `templateRef` and `viewContainer` and inject the `ViewContainerRef` and `TemplateRef<any>` types to set the properties.

Checkpoint



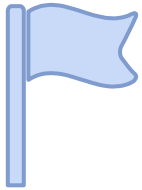
```
export class UserTypeDirective {
  private templateRef = inject(TemplateRef<any>);
  private viewContainer = inject(ViewContainerRef);
  private userType!: string;
  private userTypeClaim!: string;

  @Input() set appUserType(userType: string)
    { this.userType = userType; }

  @Input() set appUserTypeClaim(userTypeClaim: string)
    { this.userTypeClaim = userTypeClaim; }
}
```

Time to code!



1. The `UserTypeDirective` watches for changes, so add the `OnChanges` interface and `ngOnChanges(changes: SimpleChanges): void {}`.
2. Implement the `ngOnChanges()` method by creating or clearing the view if the `userTypeClaim` includes the `userType` in the string:
```
if (this.userTypeClaim.includes(this.userType)) {
  this.viewContainer.createEmbeddedView(this.templateRef);
} else { this.viewContainer.clear(); }
```
3. View changes run only when the `userTypeClaim` changes. Wrap the if/else statement creating or clearing the view with an if check on `changes['appUserTypeClaim'].previousValue !== changes['appUserTypeClaim].currentValue`.

Checkpoint



```
export class UserTypeDirective implements OnChanges {
  // prior implementation

  ngOnChanges(changes: SimpleChanges): void {
    if (changes['appUserTypeClaim'].previousValue !==
        changes['appUserTypeClaim'].currentValue) {
      if (this.userTypeClaim.includes(this.userType)) {
       this.viewContainer.createEmbeddedView(this.templateRef);
      } else { this.viewContainer.clear(); }
    }
  }
}
```

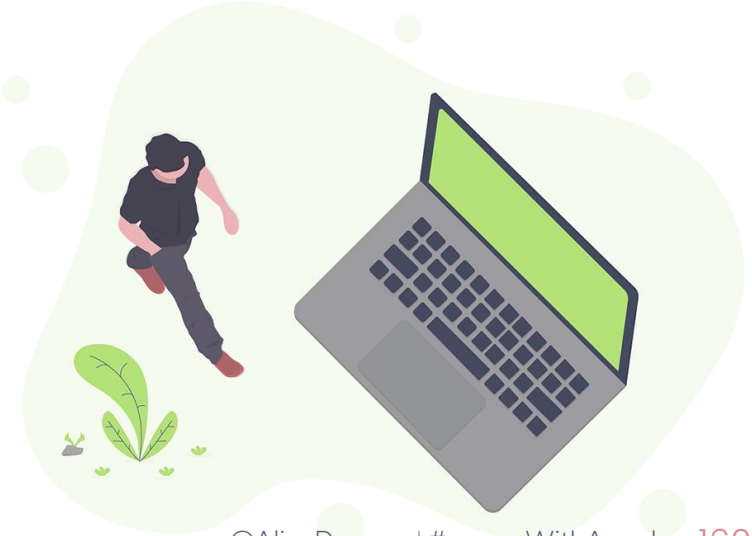
The user type defines access

Support user type

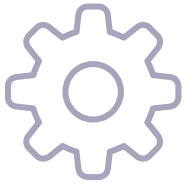
The user type should be added to the AuthService as it is a means for access control measures.

Add directive to template

Utilize the user type and the supported deal type to conditionally display deals.



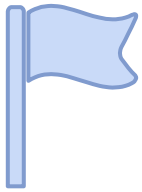
Time to code!



1. Open *app/auth.service.ts*.
2. Add a property named `userType` which returns the value from the userType claim:

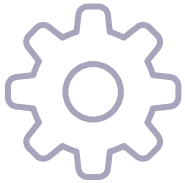
```
readonly userType = this.oidcService.userData$.pipe(
  filter(data => !!data && data.userData),
  map(data => data.userData['userType'])
);
```

Checkpoint



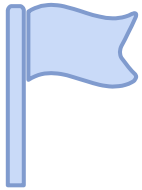
```
readonly userType = this.oidcSecurityService.userData$.pipe(
  filter(data => !!data && data.userData),
  map(data => data.userData['userType'])
);
```

Time to code!



1. Open *`home/deals.component.ts`*
2. Inject the `AuthService`.
3. Create a wrapper property for the `AuthService`'s `userType`, ensuring there's a claim value:
   ```
   userType$ = this.authService.userType.pipe(
   filter(claim => !!claim), takeUntilDestroyed());
   ```
4. In the template, update the <p> element for promo message to use the directive:
   ```
   <p *appUserType="promo.deal; claim:(userType$|async) ??
   '''' …>
   ```

Checkpoint

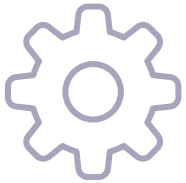


```
@Component({selector: 'app-deals',
  template: `...<ng-container *ngFor="let promo of promos">
    <p *appUserType="promo.deal; claim:(userType$|async) ?? ''"
    class="text-md">{{promo.message}}</p></ng-container>...`})
export class DealsComponent implements OnInit {
  private authService = inject(AuthService);
  public userType$ = this.authService.userType.pipe(
    filter(claim => !!claim),
    takeUntilDestroyed()
  );
  // plus existing code
}
```

Experiment!

1. Test this out by logging in as admin, you shouldn't see any deals. Log out. Change the userType claim value in Okta to "new" and try again. What do you see? How about "long-timer"?
2. Alternatively, create more users in Okta. Having trouble? Use YOPMail email service to assist with email requirements.

Compare to GitHub



alisaduncan/angular-security-workshop

1. Checkout branch `protect-views`

Recap

- Evaluate what views need to be protected and in what ways
- Utilize Angular building blocks and access control measures to prevent accidental elevation of privilege

8.
Next steps

OWASP resources
Wrap-up
Stay in touch! - Feedback/survey

Give yourself a pat on the back!

We couldn't cover everything

Insecure design

Not Angular-specific, but covering security-first design principles and considerations. Security becomes the heart of software design and development practices.

Dependency management

Keeping dependencies up-to-date, understanding how to evaluate vulnerabilities, and preventing supply chain attacks.

Content-security policies

Locally test CSPs by adding headers to *`angular.json`*'s serve options. Includes Trusted Types CSP and CSP nonce for loading resources.

Check out these great resources

Angular security docs

Angular documentation has great content. Check it out at https://angular.io/guide/security

Okta developer blog

Security and authentication learning resources

developer.okta.com/blog

OWASP Top 10

Information about each vulnerability, contributing CWEs, ways to identify problems, etc.

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

OWASP Juice Shop

Enjoy full-stack hands-on challenges? Put your skills to the test at the Juice Shop.

owasp.org/www-project-juice-shop/

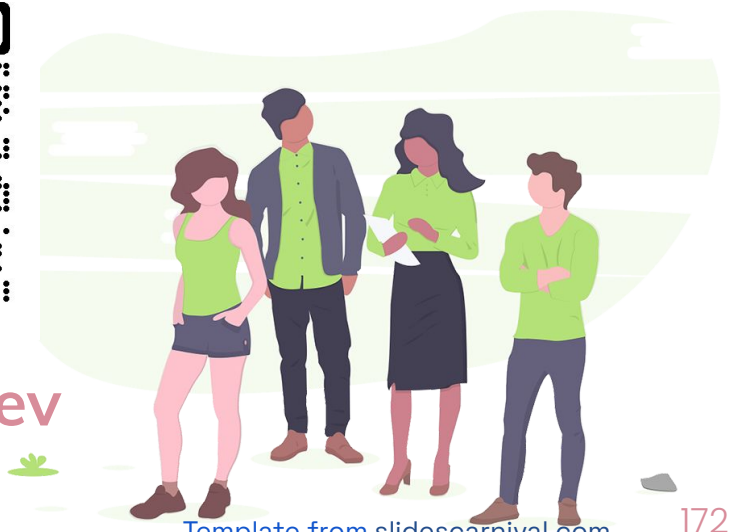
Let's be ngFriends!

@AlisaDuncan

#secureWithAngular

wic-developer-advocacy@okta.com

developer.okta.com/blog | @OktaDev



Discussion

What do you want to try next?