Angular 14 - 18

Dynamic loading of components and Feature flags

Dynamic components

- When moving from a multi-page application to a SPA, one of the **issues** that arises is the **initial load size**.
- By default, everything in an Angular application is included in a **single payload**, which means that as the application grows, the time it takes to load also increases.
- The typical way to solve this problem is by using the router to **lazy-load modules**.
- This is great when the content has a **path**, but what about situations where the components aren't part of a different path?
- For these situations we can take advantage of the Angular CLI to **split the components** into their own packages, allowing them to load only when needed.
- With introduction of Ivy compiler (https://docs.angular.lat/guide/ivy) and the new API from version 13 in advance, Angular has simplified the implementation of dynamic components.

Dynamic components

- There are several scenarios where dynamically loading a component could be useful.
- The two most obvious use cases would be for large or rarely used components.
    - In the case of a large component, it may be beneficial to let the rest of the application load and save the loading of the component for a later time.
    - Similarly, if you know that 95% of your users will never use certain components, these might be excellent candidates to load in this dynamic way to avoid initial payload impact.
- There are other less obvious cases:
    - One of them is a situation resolved by Angular.io's own pull request (PR: https://github.com/angular/angular/pull/18428).
    - Angular.io has a static HTML page with tags that are dynamically hydrated with the appropriate Angular components when needed.
    - The benefit here is that even though the page uses many Angular components, the page can initially load with the bare minimum and then incur the payload cost when actually required.

Create a dynamic component

- A dynamic component works in the same way as a **normal** one.
  - As usual, we will include it in its own module or create it as standalone component.
- The difference is that **it is not referenced** in any template or module.

```
@Component({
  selector: 'app-dynamic',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './dynamic.component.html',
  styleUrls: ['./dynamic.component.scss'],
})
export class DynamicComponent {
  show: boolean = false;
  clickCallback() {
    console.log('clickCallback: button clicked ...');
    this.show=true;
  }
}
```

```
<section>
    <h4>This is a dynamic component</h4>
    <button class="btn  btn-secondary" (click)="clickCallback()">click me</button>
    <div *ngIf="show">This is a hidden message</div>
</section>
```

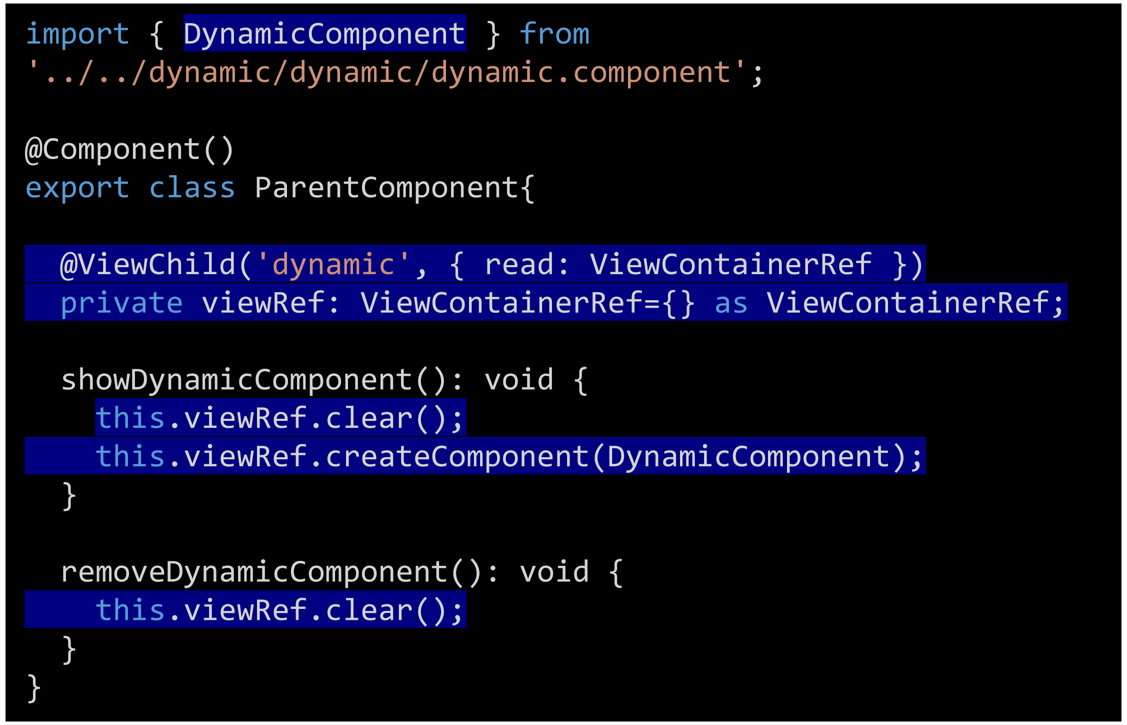
Lazy Loading a component

- Angular 13 introduced a new API for lazy loading components, that includes:
  - **@ViewChild**: configures a view query. The change detector looks for the first element or the directive matching the selector in the view DOM. If the view DOM changes, and a new child matches the selector, the property is updated.
    - https://angular.io/api/core/ViewChild
  - **ViewContainerRef**: Represents a container where one or more views can be attached to a component.
    - The **createComponent (<Component.class>)** method allows us to add adynamically a component.
    - The **clear()** method allows us to remove the component.
    - https://angular.io/api/core/ViewContainerRef
- In the template we can use `<ng-template #target></ng-template>` for injecting the dynamic component.
- There are two options for including the component from the parent:
  - **Application default bundle**: the component is transpiled in the same bundle as the rest of the application.
  - **Own bundle**: the component is transpiled in a different bundle file.

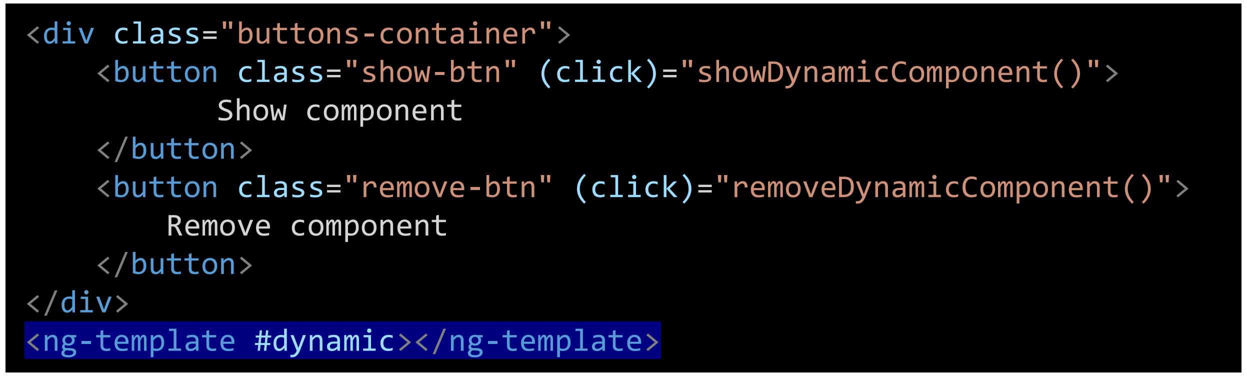
Lazy Loading a component

Application default bundle

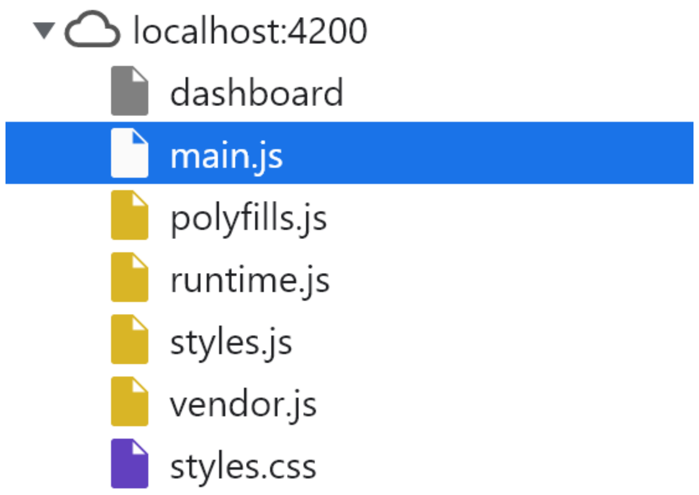
- Simply, import the component in the parent component



```
import { DynamicComponent } from
'../../dynamic/dynamic/dynamic.component';

@Component()
export class ParentComponent{

  @ViewChild('dynamic', { read: ViewContainerRef })
  private viewRef: ViewContainerRef={} as ViewContainerRef;

  showDynamicComponent(): void {
    this.viewRef.clear();
    this.viewRef.createComponent(DynamicComponent);
  }

  removeDynamicComponent(): void {
    this.viewRef.clear();
  }
}
```



```
<div class="buttons-container">
    <button class="show-btn" (click)="showDynamicComponent()">
          Show component
    </button>
    <button class="remove-btn" (click)="removeDynamicComponent()">
        Remove component
    </button>
</div>
<ng-template #dynamic></ng-template>
```



```
▼ localhost:4200
    dashboard
    main.js
    polyfills.js
    runtime.js
    styles.js
    vendor.js
    styles.css
```

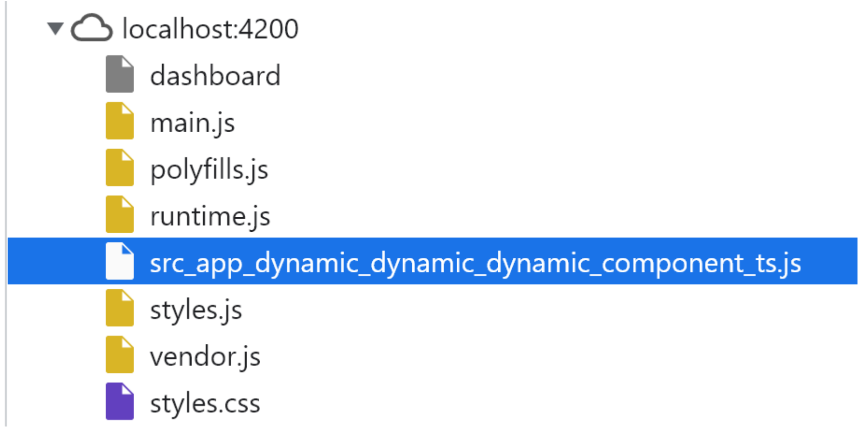
Lazy Loading a component

Own bundle

- We don’t import the component, it is loaded asyncronaly from its source file:

```
@Component()
export class DashboardComponent implements OnInit {

  @ViewChild('dynamic', { read: ViewContainerRef })
  private viewRef: ViewContainerRef={} as ViewContainerRef;

  async loadDynamicComponent() {
    const { DynamicComponent } = await
           import("../../dynamic/dynamic/dynamic.component");
    this.viewRef.clear();
    this.viewRef.createComponent(DynamicComponent);
  }

  removeDynamicComponent(): void {
    this.viewRef.clear();
  }
}
```

```
<div class="buttons-container">
    <button class="btn btn-primary“
       (click)="loadDynamicComponent()">
           Load component</button>
    <button class="btn btn-primary"
        (click)="removeDynamicComponent()">
        Remove component
    </button>
</div>
<ng-template #dynamic></ng-template>
```

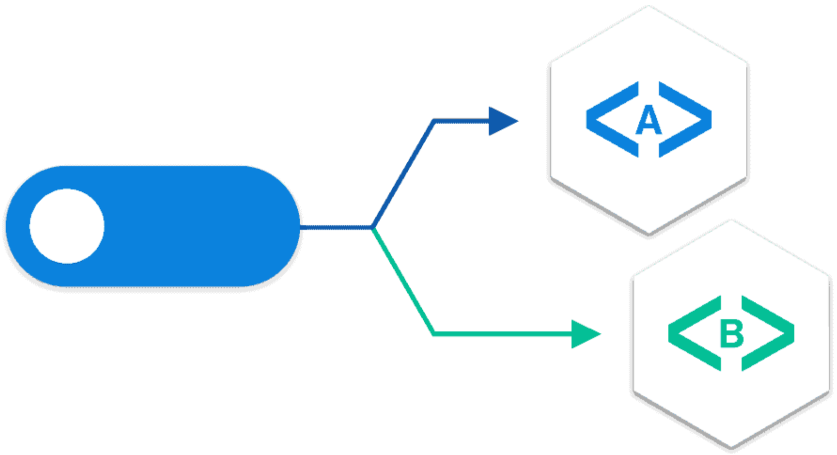


```
▼ localhost:4200
    dashboard
    main.js
    polyfills.js
    runtime.js
    src_app_dynamic_dynamic_dynamic_component_ts.js
    styles.js
    vendor.js
    styles.css
```

Let's put it into practice: Tasks/Projects App

- Create a dynamic component that allows you to preview a task in the task list when a link is clicked.

Feature flags



- Feature flags are basically a configuration for your application where we specify which features are **enabled/disabled**.
- Instead of us having to make changes in the code every time we want to enable/disable something in our application, we can make use of a **configuration file** where we specify if that feature is enabled/disabled.
  - If we are testing out a particular feature by only enabling it to a particular group of people (**A/B Testing**).
  - Or we need to **disable a feature** because it has some serious issues that would take time to fix,
- In these conditions it won't be practical to manually make changes in the code and push it every time we need to enable/disable something in the application.

Feature flags in Angular

- Implementing feature flags in Angular is easy. We will need:
  - A configuration file in the server
  - A Service to get the configuration
  - Configure APP_INITIALIZER to load it on app boostrap.
  - Route guards for controlling the access.
  - A directive for showing or hiding links or references to the feature.

Configuration File:

- Ideally this file is managed outside the application.
- Made available via an API call. In that way, we can easily update the config and the application gets the new file with ease.
- Contains a JSON object with the feature as the key and the value will be either true or false.

```
{
  “products": true,
  “newfeatures": true,
  “oldfeatures": false
}
```

Feature flags in Angular

Feature Flag service:

- It is going to manage all the logic for getting the config and also functions to check if a feature is enabled or not.

```
@Injectable()
export class FeatureFlagsService {
  config: FeatureConfig = null;
  configUrl = `${API_ROOT}/features_config`;

  constructor(private http: HttpClient) {}

  loadConfig() {
    return this.http
      .get<FeatureConfig>(this.configUrl)
      .pipe(tap((data) => (this.config = data)))
      .toPromise();
  }

  isFeatureEnabled(key: string) {
    if (this.config && has(this.config, key)) {
      return get(this.config, key, false);
    }
    return false;
  }

}
```

Feature flags in Angular

Configure APP_INITIALIZER

- We have to add our provide function so that it will call the API and loads the configuration on startup.
- For this we use a factory that will return the call the loadConfig() function in our FeatureFlagsService. And add APP_INITIALIZER in our providers array.
- When our application gets initialized, the config will be loaded in the FeatureFlagsService.

```
const featureFactory =
         (featureFlagsService: FeatureFlagsService) => () =>
           featureFlagsService.loadConfig();
...
@NgModule({
  imports: [...],
  declarations: [...],
  bootstrap: [AppComponent],
  providers: [
    {
      provide: APP_INITIALIZER,
      useFactory: featureFactory,
      deps: [FeatureFlagsService],
      multi: true,
    },
  ],
})
export class AppModule {}
```

Feature flags in Angular

Route guard

- Create a route guard to only load modules if the feature is enabled.

```
@Injectable()
export class FeatureGuard implements CanLoad {
  constructor(
    private featureFlagsService: FeatureFlagsService,
    private router: Router
  ) {}
  canLoad(route: Route,    segments: UrlSegment[]):
    | Observable<boolean | UrlTree>
    | Promise<boolean | UrlTree>
    | boolean
    | UrlTree {
    const {data: { feature },} = route;
    if (feature) {
      const isEnabled =
          this.featureFlagsService.isFeatureEnabled(feature);
      if (isEnabled) {
        return true;
      }
    }
    this.router.navigate(['/']);
    return false;
  }
}
```

Feature flags in Angular

Route guard

- Update the app-routing.module.ts file to include the guard.

```
const routes: Route[] = [
  {
    path: "new-feature",
    loadChildren: () =>
      import("./new-feature/new-feature.module")
         .then(m => m.NewFeatureModule),
    canLoad: [FeatureGuard],
    data: {
      feature: "new-feature"
    }
  },
  ...
]
```

Feature flags in Angular

Feature flag directive

- The next thing we have to do is to show the header link only when the feature is enabled. For that we will be creating a Directive, to be more precise a Structural Directive

```
import { FeatureFlagsService } from '../services/feature-flags.service';
...
@Directive({
  selector: '[appFeatureFlags]',
  standalone: true,
})
export class FeatureFlagsDirective {
  constructor(
    private tpl: TemplateRef<any>,
    private vcr: ViewContainerRef,
    private featureFlagService: FeatureFlagsService
  ) {}

  @Input() set appFeatureFlags(feature: string) {
    const isEnabled = this.featureFlagService.isFeatureEnabled(feature);
    this.vcr.clear();
    if (isEnabled) {
      this.vcr.createEmbeddedView(this.tpl);
    }
  }
}
```

```
   <nav>
...
      <ng-container *appFeatureFlags="'new-feature'">
        <a routerLink="/new-feature">New feature</a>
      </ng-container>
...
    </nav>
```

Let's put it into practice: Tasks/Projects App

- Implement feature flags configuration in the application.

Next steps

We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

Thanks!

Follow us: