Create a new project

1. Choose a place on your hard drive where the project will be stored.
2. To create a new directory `heroes-ui` with an empty project inside execute in the terminal

```
npx -y @angular/cli@22 new heroes-ui --style=css --ssr=false --zoneless=true
                                     --ai-config=none
```

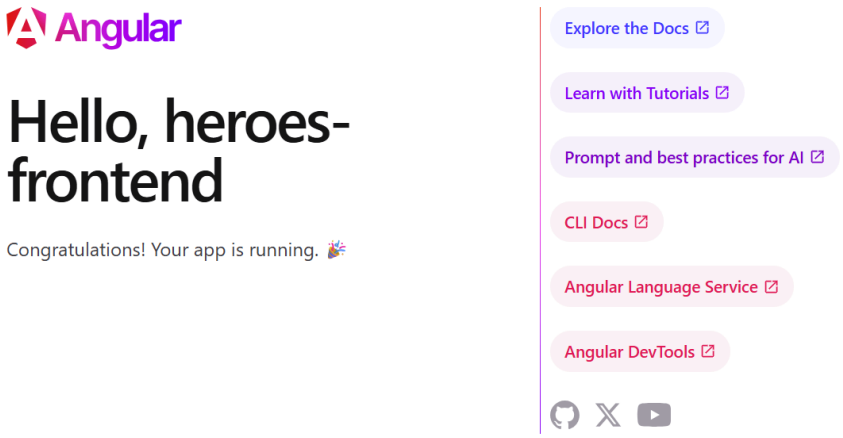
3. `new` will create the standard project structure inside, and will download and install required Angular dependencies which can take some time
    a. Additional options are not required, but will create a project with a specific configuration so you wouldn't need to do it manually
4. **[optional]** on the page https://angular.dev/cli/new you can find details on the used and many other parameters

Start the project

1. Go to the created directory and execute

```
ng serve --open
```

2. The command will
    a. build the application
    b. start the development server
    c. watch the source files (so if you change them, the server will automatically rebuild the application and will make the update available)
    d. open the main page in your default browser
3. In your browser, you should see



Initial changes

1. Open the project folder in any IDE, open `index.html` and change `<title>HeroesUi<title>` to `<title>Tour of Heroes</title>`.

    a. You just changed the only real webpage your application has - it will affect every component you will create as all of them will be placed into this webpage.
2. Open `src/app/app.ts` and replace the value of the signal `title` from `heroes-ui` to `Tour of Heroes`.
3. Delete all content from `src/app/app.html`, put `<h1>{{ title }}</h1>` instead, and check the browser.
    a. You just used interpolation binding to inject the value of the variable `title` in the TypeScript class into the HTML template that was used to generate a webpage you see in your browser.
4. Open src/app/app.css and insert

```
/* Application-wide Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[type="text"], button {
  color: #333;
  font-family: Cambria, Georgia, serif;
}
button {
  background-color: #eee;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  color: black;
  font-size: 1.2rem;
  padding: 1rem;
  margin-right: 1rem;
  margin-bottom: 1rem;
  margin-top: 1rem;
}
button:hover {
  background-color: black;
  color: white;
}
button:disabled {
  background-color: #eee;
  color: #aaa;
  cursor: auto;
}
```

```
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

5. Check the browser.
    a. You just added styles to your HTML tags specifying how they should look on your website.
    b. You could notice that some of them have “modifiers” such as `:hover` or `:disabled`. They change styling for some specific cases. For example, `:hover` defines how the tag will look like if there is a mouse pointer over it, and `:disabled` defines how the tag will look like if it’s disabled (in our case - a user cannot click on a button).

List of heroes

1. First of all, we need a Hero model in our project, so create a file `src/app/hero.ts` and add the following content there

```
export interface Hero {
  id: number;
  name: string;
}
```

2. As our frontend does not communicate with the Heroes API yet, we temporarily need to create heroes that we can use to check if our frontend works properly. Create a file `src/app/mock-heroes.ts` and add the following content there

```
import { Hero } from './hero';

export const HEROES: Hero[] = [
  { id: 12, name: 'Dr. Nice' },
  { id: 13, name: 'Bombasto' },
  { id: 14, name: 'Celeritas' },
  { id: 15, name: 'Magneta' },
  { id: 16, name: 'RubberMan' },
  { id: 17, name: 'Dynama' },
  { id: 18, name: 'Dr. IQ' },
  { id: 19, name: 'Magma' },
  { id: 20, name: 'Tornado' }
];
```

3. Now we need to create a component that will show them, so go to the root folder of the frontend project and execute

```
ng g component heroes
```

4. You have a new folder `src/app/heroes` and several files inside - a TypeScript class, a HTML template, CSS styles and .spec.ts that is used to store unit tests (you can ignore this one).

a. In the TypeScript class, you have a decorator `@Component` with several parameters: `selector` - a new tag you can use to put the created heroes component into, `templateUrl` - HTML template name, `styleUrl` - a file with styles for the component exclusively.

5. Add `<app-heroes></app-heroes>` to `src/app/app.html`.
   a. Check the browser - now it shows the content of `src/app/heroes/heroes.html` as you added the heroes components to the main page.
6. Now let's add our fake heroes to the heroes components so we would be able to show them by adding `import { HEROES } from '../mock-heroes';` to `src/app/heroes/heroes.ts` and update the class body so it would look like

```
export class Heroes {

  heroes: Hero[] = HEROES;
  selectedHero?: Hero;

  onSelect(hero: Hero): void {
    this.selectedHero = hero;
  }

}
```

   a. the variable `heroes` is becoming equal to `HEROES` that contains our fake heroes at the moment the class is created
   b. the variable `selectedHero` contains a hero object (empty by default), and `?` indicates the variable can be null
   c. when the function `onSelect` is called, it takes the `hero` parameter and assigns it to the `selectedHero` variable (we will use it in the next step)

7. To actually show those heroes on the page, replace the existing content of `src/app/heroes/heroes.html` with

```
<h2>My Heroes</h2>
<ul class=”heroes”>
  @for (hero of heroes; track hero.id) {
    <li>
      <button [class.selected]="hero === selectedHero" type="button"
(click)="onSelect(hero)">
        <span class="badge">{{ hero.id }}</span>
        <span class="name">{{ hero.name }}</span>
      </button>
    </li>
  }
</ul>
```

   a. `@for` - iterates through the list of heroes stored in the variable `heroes` in the class of the heroes component
   b. `track` - tells Angular how to establish uniqueness of the objects in the list, it helps it to redraw only items that have been changed instead of redrawing everything

c. when a user clicks on a button for some specific hero, the function `onSelect` from the heroes components is called, and the corresponding hero object is passed as a parameter - we use event binding for this
d. we want to highlight the currently chosen hero with a different styling, so we use class binding (subset of property binding) - every time the expression `hero === selectedHero` is true, the button will have the CSS class `.selected`, and it allows us to display this specific element differently through CSS customization
e. the final behaviour is - a user clicks on the button for the hero H, the function `onSelect` from the class is called with the object H as a parameter, `selectedHero` is becoming equal to hero, as the expression `hero === selectedHero` is now true, the button for the hero H obtains the CSS class `.selected` which changes how this button looks
f. notice that we don't put `html` and `body` tags into `src/app/heroes/heroes.html` - Angular will inject the content of the file into `src/app/app.html` which in injected into `src/main.html` which already has these tags

8. Now it's time to polish the visual part. Update the file `src/app/heroes/heroes.css` with the following content

```
/* Heroes' private CSS styles */
.heroes {
  margin: 0 0 2em 0;
  list-style-type: none;
  padding: 0;
  width: 15em;
}

.heroes li {
  display: flex;
}

.heroes button {
  flex: 1;
  cursor: pointer;
  position: relative;
  left: 0;
  background-color: #EEE;
  margin: .5em;
  padding: 0;
  border-radius: 4px;
  display: flex;
  align-items: center;
  height: 2.5em;
  border-width: 0rem;
}

.heroes button:hover {
  color: #2c3a41;
  background-color: #e6e6e6;
  left: .1em;
```

```
}

.heroes button:active {
  background-color: #525252;
  color: #fafafa;
}

.heroes button.selected {
  background-color: black;
  color: white;
}

.heroes button.selected:hover {
  background-color: #505050;
  color: white;
}

.heroes button.selected:active {
  background-color: black;
  color: white;
}

.heroes .badge {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100%;
  padding: 0 0.7em;
  font-size: small;
  color: white;
  background-color: #405061;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}
```

9. If we select a hero, we want to show some additional details about that specific hero. To do it, add the following content to `src/app/heroes/heroes.html`

```
@if(selectedHero) {
  <h2>{{ selectedHero.name | uppercase }} Details</h2>
  <div>Id: {{ selectedHero.id }}</div>
  <div>
    <label for="hero-name">Hero name: </label>
    <input id="hero-name" [(ngModel)]="selectedHero.name"
placeholder="name">
  </div>
}
```

   a. `@if` - the content of the block will be shown only if the condition is true

    b. `uppercase` - a “pipe” (operation) that is applied to the value you receive through interpolation binding, we can chain multiple operations in one interpolation
    c. `[(ngModel)]` - connects the input field with the variable selectedHero in the component class using two-way binding
10. Unfortunately, the uppercase operation doesn’t work as is, so we need to import it. To do it, add `import { UpperCasePipe } from '@angular/common';` to `src/app/heroes/heroes.ts` imports and replace `imports: [],` with `imports: [ UpperCasePipe ],` in the decorator.
11. `[(ngModel)]` doesn’t work either, it needs functionality from `FormsModule`. To add it, add `import { FormsModule } from '@angular/forms';` to `src/app/heroes/heroes.ts` imports and add `FormsModule` to `imports: [ UpperCasePipe ],` in the class decorator.
12. If you switch to the browser and click on any of the heroes, it will become “selected” (`selectedHero` variable in the component class), and it means the condition `selectedHero` in `@for(selectedHero)` will become true which will force Angular to draw the content of the if block. If you click on another hero, `selectedHero` will be updated, and the content of the if block will consequently be updated for the newly selected hero
13. If you select a hero, you will be able to edit their name, and two-way binding will immediately propagate the change into the hero object in the component class. As `selectedHero` is just another pointer to one of the heroes in the list, the name of the selected hero in the list will be immediately updated and that update will be reflected on the corresponding button on the website
    a. As the hero list exists in memory and is created each time the application is loaded, if you refresh the page, all your changes will disappear

Feature component

While the current frontend application works just fine, we have two different functionalities - a list of heroes and hero details - inside of one component which goes against the separate responsibility principle and is not maintainable in the long run. If we add more and more functionality, we will have quite a complex and messy codebase which will be hard to maintain. Let’s split a list and details and create a separate component for showing hero details.

1. Create a new component by going to the root folder of the application and executing

```
ng g component hero-detail
```

2. Remove the if block from `src/app/heroes/heroes.html` and replace the content of `src/app/hero-detail/hero-detail.html` with the following

```
@if(hero) {
  <h2>{{ hero.name | uppercase }} Details</h2>
  <div><span>Id: </span>{{ hero.id }}</div>
  <div>
    <label for="hero-name">Hero name: </label>
    <input id="hero-name" [(ngModel)]="hero.name" placeholder="name">
  </div>
}
```

a. Note that we replaced `selectedHero` with just `hero`. This component is more generic and can show any hero that will be passed to it, so it's not bound to a specific variable in another component.

3. To add some life to new hero detail component, add `import { Hero } from '../hero';` to `src/app/hero-detail/hero-detail.ts` imports, replace `import { Component } from '@angular/core';` with `import { Component, Input } from '@angular/core';` and add `@Input() hero?: Hero;` to the class body so it would look like

```
export class HeroDetail {

  @Input() hero?: Hero;

}
```

   a. @Input() decorator tells Angular that the value of the property will be passed from the outside (some other component). Without that decorator it will be impossible to do.

4. Do not forget to import `UpperCasePipe` from `@angular/common` and `FormsModule` from `@angular/forms` to `src/app/hero-detail/hero-detail.ts` as we did before for `src/app/heroes/heroes.ts` and add them into the imports property inside of the class decorator.
5. As we want to use the hero detail component inside of the heroes component, we need to import it. So, add `import { HeroDetail } from '../hero-detail/hero-detail';` to the heroes component imports and `HeroDetail` to the class decorator. The final version of the heroes component's class should look like this

```
import { Component } from '@angular/core';
import { HeroDetail } from '../hero-detail/hero-detail';
import { Hero } from '../hero';
import { HEROES } from '../mock-heroes';

@Component({
  selector: 'app-heroes',
  imports: [ HeroDetail ],
  templateUrl: './heroes.html',
  styleUrl: './heroes.css',
})
export class Heroes {

  heroes: Hero[] = HEROES;
  selectedHero?: Hero;

  onSelect(hero: Hero): void {
    this.selectedHero = hero;
  }

}
```

   a. Notice that we removed `UpperCasePipe` and `FormsModule` - we don't need them in the heroes component anymore.

6. Open `src/app/heroes/heroes.html` and add

```
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
```

   a. `app-hero-detail` is the tag defined in the `@Component` decorator of the hero detail class, so instead of using the code showing hero details directly in the heroes component, we extract it into a separate component and pass the currently selected hero into its property letting it handle drawing. As a result, if we want to show hero details somewhere else, we don't need to duplicate the code, we can just inject the component and reuse it.
   b. Notice that `[hero]="selectedHero"` is just property binding, and it looks contr-intuitive as we have learnt that we can use property binding to pass values from the variables of a class to tag's properties in HTML. It's correct, but when we compose components into parent-child relationships (heroes is a parent and hero detail is a child), we can use property binding to push values from a parent to a child. Simplifying, we can see it as

```
const child = new HeroDetailComponent();
child.hero = this.selectedHero;
```

7. Switch to the browser and make sure that from the user point of view everything works as before.

Add a service

Right now, we take a list of fake heroes and put it into the heroes component directly. In the future, we want to fetch the data of real heroes from Heroes API, so let's separate data presenting and data obtaining.

1. Go to the root folder of the project and execute

```
ng g service hero --type=service
```

   a. Starting from Angular 22, `@Injectable` decorator is replaced by `@Service` decorator when a service is generated (but you still can see `@Injectable` in many articles online). Both can be used, but `@Service` will work just fine in the majority of the situations and doesn't require additional parameters (such as `providedId: 'root'`)
2. To make the service be able to provide heroes to the components, add imports `import { Hero } from './hero';` and `import { HEROES } from './mock-heroes';` to `src/app/hero.service.ts` and add the function to the class body

```
getHeroes(): Hero[] {
  return HEROES;
}
```

Let's now use the hero service to get heroes in the heroes component instead of using the set of fake heroes directly.

3. Replace the import `import { HEROES } from '../mock-heroes';` with `import { HeroService } from '../hero.service';` in `src/app/heroes/heroes.ts` and replace `heroes = HEROES;` with `heroes: Hero[] = [];` in the same class - now by default the list of heroes will be empty when the application is loaded.

4. Replace `import { Component } from '@angular/core';` with `import { Component, inject } from '@angular/core';` in `src/app/heroes/heroes.ts` and inject the hero service into the heroes component by adding a new variable

```
private heroService: HeroService = inject(HeroService);
```

   a. In previous versions of Angular, you would use a standard constructor (and you still can), but in new versions `inject` is the preferred way.

5. We need to obtain heroes from the service, so we need to add a method for this to `src/app.heroes/heroes.ts`

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```

6. Finally, we need to trigger heroes fetching from the service when the application is loaded. In theory, we could do it in the constructor, but it's not the best practice as we cannot guarantee that when the component class is constructed the rest of the application (including the service we want to call) is initialized and ready to be used. To handle this problem, we can use the standard method `ngOnInit()` which is called when the component is fully initialized with all its dependencies. Add this method to `src/app/heroes/heroes.ts`

```
ngOnInit(): void {
  this.getHeroes();
}
```

   a. As a result, when the component class constructor is called, the service is injected into the class, after that when the component is fully initialized with its dependencies (including the service) ngOnInit() is triggered and the hero service is called to get the set of fake heroes and assign them to the heroes variable.
7. Switch to the browser and check that everything works as before.

At the moment, we use just a predefined fixed list of heroes, but in future we want to fetch them from our backend via standard `HttpClient` provided by Angular. However, `HttpClient` produces not just data but `Observables` with data. It means our current code is not compatible with these future changes. Let's fix it by introducing observables.

8. First of all, add `import { Observable, of } from 'rxjs';` to `src/app/heroes.service.ts` and update `getHeroes()` function so it would return an observable

```
getHeroes(): Observable<Hero[]> {
    return of(HEROES);
}
```

   a. `of(HEROES)` transforms `Hero[]` into an observable with a single value which is an array of heroes - `Observable<Hero[]>`

9. Now the service returns `Observable<Hero[]>`, but the component expects to receive just `Hero[]`. To fix the situation replace `getHeroes()` method in `src/app/heroes/heroes.ts` with

```
getHeroes(): void {
  this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes);
}
```

10. Switch to the browser and check that everything works as before.