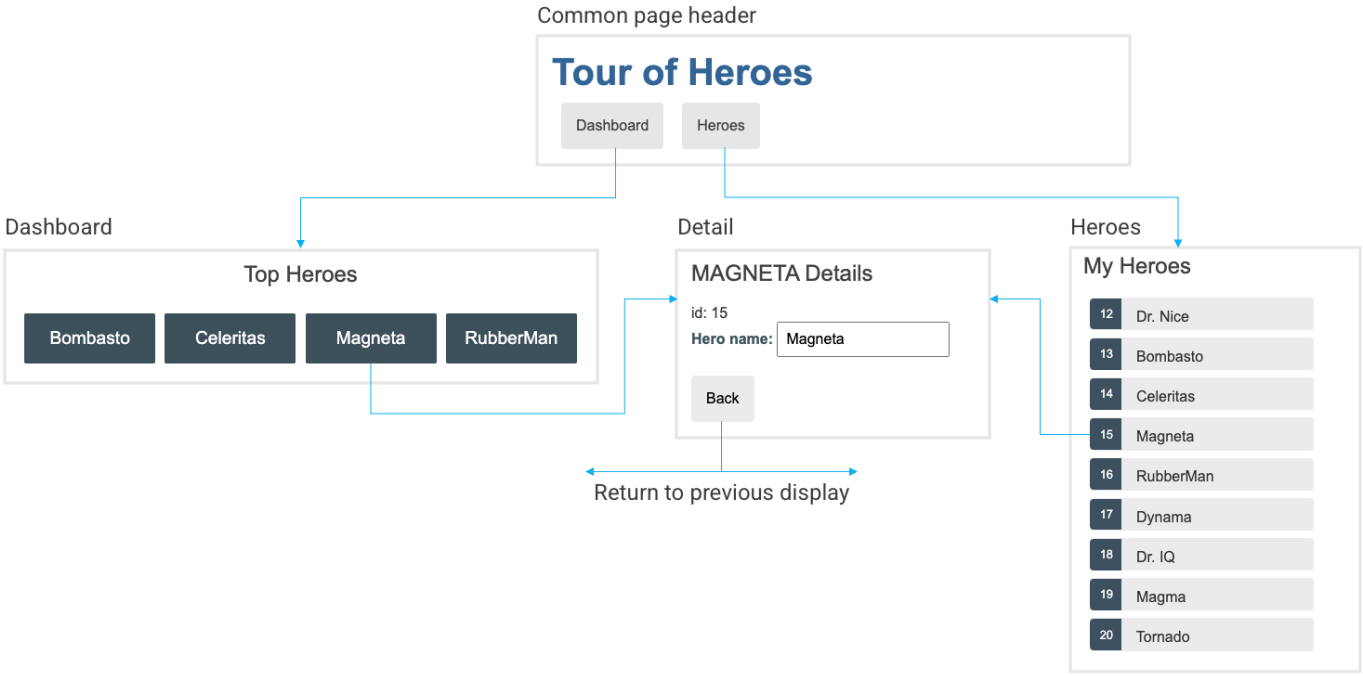
Add navigation

We have everything on one screen, but the real applications are more complex than this, so let's add a bit more functionality and split all the functionality to several separate screens. When we are done, users will be able to navigate through our application like this



As we can see from the screenshot, we will need several pages - 1) a dashboard page, 2) a hero list page, and 3) a hero details page. In addition, each of these should have a separate URL so users would be able to share links leading directly where they need. To make it even more convenient, we should be able to give links to specific heroes, so the URL for Magneta details and the URL for Bombasto details would be different. Don't forget that all these pages are "virtual" - we still build a single page application, so we have just one real HTML page, and all the content will be switched inside of it depending on the current URL.

Let's prepare our application for routing

1. The application will show the required content dynamically, so we don't need the Heroes component imported to the app component anymore - delete `import { Heroes } from "./heroes/heroes";` from `src/app/app.ts` and replace `imports: [RouterOutlet, Heroes],` with `imports: [RouterOutlet],` in the same file, and remove `<app-heroes></app-heroes>` from `src/app/app.html`
2. As we need to navigate through our application somehow, add `<router-outlet></router-outlet>` to `src/app/app.html` - from now on all the dynamic content managed by Angular navigation will be shown instead of the `router-outlet` tag.
3. To let Angular know the mapping between pages and URLs, replace the content of `src/app/app.routes.ts` with

```
import { Routes } from '@angular/router';
import { HeroDetail } from './hero-detail/hero-detail';
import { Heroes } from './heroes/heroes';

export const routes: Routes = [
  { path: '', redirectTo: '/heroes', pathMatch: 'full' },
  { path: 'heroes', component: Heroes},
  { path: 'heroes/:id', component: HeroDetail},
];
```

4. Now if a user opens `http://localhost:4200/heroes`, they will see our list of heroes. But as we don't want a user to guess what URLs our application has, we added a default redirect - if just `http://localhost:4200` is opened, the application automatically sends a user to `http://localhost:4200/heroes`. This approach can be used if you do not have a main page that you want to show to a user. We also have `http://localhost:4200/heroes/:id` path in our routing, but as `HeroDetail` component doesn't see `id` as a parameter, it won't work. Let's fix it.
5. As we want to show hero information on a separate page, it shouldn't exist on the page with a list of heroes. Delete `<app-hero-detail [hero]="selectedHero"></app-hero-detail>` from `src/app/heroes/heroes.html` and replace the content of the `@for` loop with

```
<li>
  <a [routerLink]="['/heroes', hero.id]">
    <span class="badge">{{ hero.id }}</span>
    <span class="name">{{ hero.name }}</span>
  </a>
</li>
```

6. As we do not use `selectedHero` in `src/app/heroes/heroes.ts` anymore, delete the variable, the `onSelect(hero: Hero)` method and imports of `HeroDetail`.
7. The functionality is there, but the styling is completely broken as in the previous version we used buttons but now it's links. Replace the content of `src/app/heroes/heroes.css` with

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

.heroes li {
  position: relative;
  cursor: pointer;
}

.heroes li:hover {
  left: .1em;
}
```

```
.heroes a {
  color: #333;
  text-decoration: none;
  background-color: #EEE;
  margin: .5em;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px;
  display: flex;
  align-items: center;
  width: 100%;
}

.heroes a:hover {
  color: #2c3a41;
  background-color: #e6e6e6;
}

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

.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #405061;
  line-height: 1em;
  position: relative;
  left: -1px;
  height: 1.8em;
  min-width: 16px;
  text-align: right;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}
```

8. We have a working list of links to individual hero pages, but those pages are not reachable. Let's fix it.
9. Replace `import { Component, Input } from '@angular/core';` with `import { Component, Input, inject } from '@angular/core';` in `src/app/hero-detail/hero-detail.ts`, add `import { ActivatedRoute } from '@angular/router';`, and add the variable `private route: ActivatedRoute = inject(ActivatedRoute);` to the class. `ActivatedRoute` allows us to get access to the current URL and its query parameters, so we will be able to parse the URL and find the hero id.
10. We will need to obtain a hero with a specific id, so let's add the following method to `src/app/hero.service.ts`

```
getHero(id: number): Observable<Hero> {
  const hero = HEROES.find(hero => hero.id === id)!;
  return of(hero);
}
```

11. Add `import { HeroService } from '../hero.service';` to `src/app/hero-detail/hero-detail.ts`, and inject it by `private heroService: HeroService = inject(HeroService);`
12. Every time we open an URL for a hero with a specific id, we need to load the corresponding hero, so add the following methods to `src/app/hero-detail/hero-detail.ts` to do it

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

getHero(): void {
  const id = Number(this.route.snapshot.paramMap.get('id'));
  this.heroService.getHero(id)
    .subscribe(hero => this.hero = hero);
}
```

13. Switch to the browser and try to click on different heroes in the list.
14. Let's try to create a "back" button that will emulate browser behaviour. For `src/app/hero-detail/hero-detail.ts`, add `Location` to `import { UpperCasePipe } from '@angular/common';`, inject it with `private location: Location = inject(Location);` and add the function

```
goBack(): void {
  this.location.back();
}
```

15. Add `<button type="button" (click)="goBack()">go back</button>` to the very bottom of `src/app/hero-detail/hero-detail.html`
16. Switch to the browser and check the added "back" button.
17. To create a dashboard component, go to the root directory of the project in your terminal and execute

```
ng g component dashboard
```

18. Add the route `{ path: 'dashboard', component: Dashboard},` to `src/app/app.routes.ts` and import the component `import { Dashboard } from './dashboard/dashboard';`
19. For `src/app/dashboard/dashboard.ts`, import `RouterLink` to the file imports and to the component imports, import `HeroService` and inject it, add a variable for a list of heroes and methods for obtaining a subset of heroes from `HeroService`. Try to implement it by yourself before checking the expected result. The final version of the class should look like

```
import { Component, inject } from '@angular/core';
import { RouterLink } from '@angular/router';
```

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

@Component({
  selector: 'app-dashboard',
  imports: [ RouterLink ],
  templateUrl: './dashboard.html',
  styleUrl: './dashboard.css',
})
export class Dashboard {

  private heroService: HeroService = inject(HeroService);

  protected heroes: Hero[] = [];

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

  getHeroes(): void {
    this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes.slice(1, 5));
  }

}
```

20. Replace the content of `src/app/dashboard/dashboard.html` with

```
<h2>Top Heroes</h2>
<div class="heroes">
  @for (hero of heroes; track hero.id) {
    <a [routerLink]="['/heroes', hero.id]">{{ hero.name }}</a>
  }
</div>
```

21. To style the component put the following CSS to `src/app/dashboard/dashboard.css`

```
/* Dashboard's private CSS styles */
h2 {
  text-align: center;
}

.heroes {
  padding: 0;
  margin: auto;
  max-width: 1000px;
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: space-around;
```

```
  align-content: flex-start;
  align-items: flex-start;
}

a {
  background-color: #3f525c;
  border-radius: 2px;
  padding: 1rem;
  font-size: 1.2rem;
  text-decoration: none;
  display: inline-block;
  color: #fff;
  text-align: center;
  width: 100%;
  min-width: 70px;
  margin: .5rem auto;
  box-sizing: border-box;
  order: 0;
  flex: 0 1 auto;
  align-self: auto;
}

@media (min-width: 600px) {
  a {
    width: 18%;
    box-sizing: content-box;
  }
}

a:hover {
  background-color: #000;
}
```

22. Open `http://localhost:4200/dashboard` in your browser and test the result.
23. We have a dashboard and a list of heroes, but to navigate between them, we need to use URLs. To add navigation, add `RouterLink` to the file import and the component import in `src/app/app.ts`, and in `src/app/app.html` between the title and router-outlet add the following code

```
<nav>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/dashboard">Dashboard</a>
</nav>
```

24. To visually enhance the navigation bar, add the following CSS to `src/app/app.css`

```
nav {
  display: flex;
  gap: 0.5rem;
  margin-top: 10px;
```

```
}

nav a {
  padding: 1rem;
  text-decoration: none;
  display: inline-block;
  background-color: #e8e8e8;
  color: #3d3d3d;
  border-radius: 4px;
}

nav a:hover {
  color: white;
  background-color: #42545C;
}

nav a:active {
  background-color: black;
}
```

Server Integration

We have been building a frontend that shows and manipulates ephemeral data stored in its variable only. When you restart the application, all the changes are lost, and to make them permanent you need to change the codebase which is not maintainable outside of a toy exercise. To add persistence and to be able see changes other users make, we need to fetch the data from our backend and propagate our changes there. Our backend provides REST API over HTTP, and fortunately Angular has HttpClient that is designed specifically for communicating with such APIs. To test further changes, you need to have Hero API running as our frontend will try to interact with its endpoints to read and write data.

1. Import `HttpClient` to `src/app/hero.service.ts` by `import { HttpClient } from '@angular/common/http';` and inject it `private httpClient: HttpClient = inject(HttpClient);` (don't forget to import `inject` - `import { Service, inject } from '@angular/core';`)
2. Replace `getHeroes()` method in `src/app/hero.service.ts` with

```
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(`http://localhost:8080/heroes`);
}
```

3. Please, pay attention to the quote symbols in all URLs inside of the service - they allow you to inject parameters into strings (as in the next item). If you use another quotation symbol, injection will not work.
4. Replace getHero(id: number) method in src/app/hero.service.ts with

```
getHero(id: number): Observable<Hero> {
  return this.http.get<Hero>(`http://localhost:8080/heroes/${id}`);
}
```

5. Remove all unused imports.
6. Angular doesn't trace changes happening within callbacks directly, so we need to use signals. In `src/app/heroes/heroes.ts`, replace `import { Component, inject } from '@angular/core';` with `import { Component, inject, WritableSignal, signal } from '@angular/core';`, and `protected heroes: Hero[] = [];` with `protected heroes: WritableSignal<Hero[]> = signal<Hero[]>([]);` and `getHeroes()` method with

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

7. As a result, our heroes variable in `src/app/heroes/heroes.ts` is now not an array, but a signal, and it means we cannot just iterate over it in the HTML-template. In `src/app/heroes/heroes.html`, we need to replace `heroes` in `@for` loop with `heroes()` to get the value of the signal.
8. Check your browser and make sure the list of heroes works properly. Nothing else will work, though.
9. Fix the dashboard by introducing the same changes you made for the list of heroes (but keep the slicing).
10. Let's do the same for `src/app/hero-detail/hero-detail.ts` - import `WritableSignal` and `signal`, replace the `hero` variable with `@Input() protected hero: WritableSignal<Hero | undefined> = signal<Hero | undefined>(undefined);` and update the `getHero()` method accordingly.
11. To adjust `src/app/hero-detail/hero-detail.html`, replace `@if(hero)` with `@if(hero(); as hero)`
12. Check your browser and make sure data fetching works properly everywhere.
13. The problem is while we can get hero data from a server, if you change it and go to another screen or refresh the page, changes are lost. They exist only in the variable inside of the component. Every time we reload it, the changed data is lost and we get the server data instead. We need to have a way to send the changed data to the server so the next reload would fetch it.
14. Update the import in `src/app/hero.service.ts` from `import { HttpClient } from '@angular/common/http';` to `import { HttpClient, HttpHeaders } from '@angular/common/http';`, and add the variable

```
private httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json'
  })
};
```

15. To `src/app/hero.service.ts`, add

```
updateHero(hero: Hero): Observable<Hero> {
  return
this.http.put<Hero>(`http://localhost:8080/heroes/${hero.id}`, hero,
this.httpOptions);
}
```

16. To `src/app/hero-detail/hero-detail.ts`, add

```
save(): void {
  const hero: Hero | undefined = this.hero();
  if (hero) {
    this.heroService.updateHero(hero)
      .subscribe(() => this.goBack());
  }
}
```

17. Note that we create an internal variable `hero` instead of using `this.hero()` in the check and in the passing to `updateHero()` directly. The catch is that while `WritableSignal` is a variable, getting a value of the signal is a function call, and we cannot guarantee two different calls produce the same result (it can be a hero on the first call and undefined on the second one).
18. Add `<button type="button" (click)="save()">save</button>` to `src/app/hero-detail/hero-detail.html` and try to change some hero's name and save it in your browser.
19. Now let's try to implement hero creation.
20. Add the following method to `src/app/hero.service.ts`

```
createHero(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(`http://localhost:8080/heroes`, hero,
this.httpOptions);
}
```

21. Add the following method to `src/app/heroes/heroes.ts`

```
create(name: string): void {
  name = name.trim();
  if (!name)
    return;
  this.heroService.createHero({ name } as Hero)
    .subscribe(hero => {
      this.heroes.update(heroes => [...heroes, hero]);
    })
}
```

22. Add the form right after the header in `src/app/heroes/heroes.ts`

```
<div>
  <label for="new-hero">Hero name: </label>
  <input id="new-hero" #heroName />
  <button type="button" class="add-button"
(click)="create(heroName.value); heroName.value=''">
    Add hero
  </button>
</div>
```

23. Finally, add styling to `src/app/heroes/heroes.css`

```
.add-button {
  display: inline-block;
  padding: 0.5rem 1.5rem;
  font-size: 1rem;
  color: #333;
  background-color: #eee;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  margin-bottom: 2rem;
  margin-left: 1rem;
}

.add-button:hover {
  color: white;
  background-color: #42545C;
}
```

24. Switch to your browser and check that hero creation is working.
25. Let's say we don't like the hero we created and want to delete it. Let's add such functionality.
26. For `src/app/hero.service.ts`, add the method

```
deleteHero(id: number): Observable<void> {
  return this.http.delete<void>(`http://localhost:8080/heroes/${id}`,
this.httpOptions);
}
```

27. Note, that the method returns an `Observable` with nothing - we still need to interact with the server asynchronously, but the server doesn't return any useful body in the response.
28. To call this method, in `src/app/heroes/heroes.ts` add

```
delete(hero: Hero): void {
  this.heroService.deleteHero(hero.id).subscribe(
    () => this.heroes.update(heroes => heroes.filter(h => h.id !==
hero.id));
  );
}
```

29. Note that normally we would have a normal variable name in `.subscribe()` that would contain a result of function execution, but in our case we use `()`. It's a convention for cases where there is no data, but we still need to put something there due to language's syntax requirements. In case we had some data but chose to ignore it, we would use `_`.
30. In `src/app/heroes/heroes.html`, right after `<a>...</a>`, but before `</li>`, to call the component method, put

```
<button type="button" class="delete-button" title="delete hero"
(click)="delete(hero)">x</button>
```

31. In `src/app/heroes/heroes.css`, replace `.heroes li {...}` block with

```
.heroes li {
  position: relative;
  display: flex;
  align-items: center;
  cursor: pointer;
}
```

32. In `src/app/heroes/heroes.css`, replace `.heroes a {...}` block with

```
.heroes a {
  color: #333;
  text-decoration: none;
  background-color: #EEE;
  margin: .5em 0;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px 0 0 4px;
  display: flex;
  align-items: center;
  flex: 1;
}
```

33. In `src/app/heroes/heroes.css`, add

```
.delete-button {
  height: 2.2em;
  width: 2.2em;

  margin: .5em 0;
  padding: 0;

  border: none;
  border-left: 1px solid #ccc;
  border-radius: 0 4px 4px 0;

  background-color: #EEE;
  color: #525252;

  font-size: 1rem;
  cursor: pointer;
}

.delete-button:hover {
  background-color: #525252;
  color: white;
}
```

34. Switch to your browser and check that everything works as intended.
35. There is one last endpoint we need to utilize - search.
36. First, add the following method to `src/app/hero.service.ts`

```
searchHeroes(text: string): Observable<Hero[]> {
  return
this.http.get<Hero[]>(`http://localhost:8080/heroes?name=${text}`,
this.httpOptions);
}
```

37. Add a new component by executing the following command in the application root directory

```
ng g component hero-search
```

38. Replace the content of `src/app/hero-search/hero-search.ts` with the following content

```
import { Component, output } from '@angular/core';

@Component({
  selector: 'app-hero-search',
  imports: [],
  templateUrl: './hero-search.html',
  styleUrl: './hero-search.css',
})
export class HeroSearch {

  searchChange = output<string>();

  onInput(event: Event) {
    const input = event.target as HTMLInputElement;
    this.searchChange.emit(input.value);
  }

}
```

39. Replace the content of `src/app/hero-search/hero-search.html` with the following content

```
<div id="search-component">
  <label for="search-box">Hero Search</label>
  <input type="search" placeholder="Hero's name" id="search-box"
(input)="onInput($event)" />
</div>
```

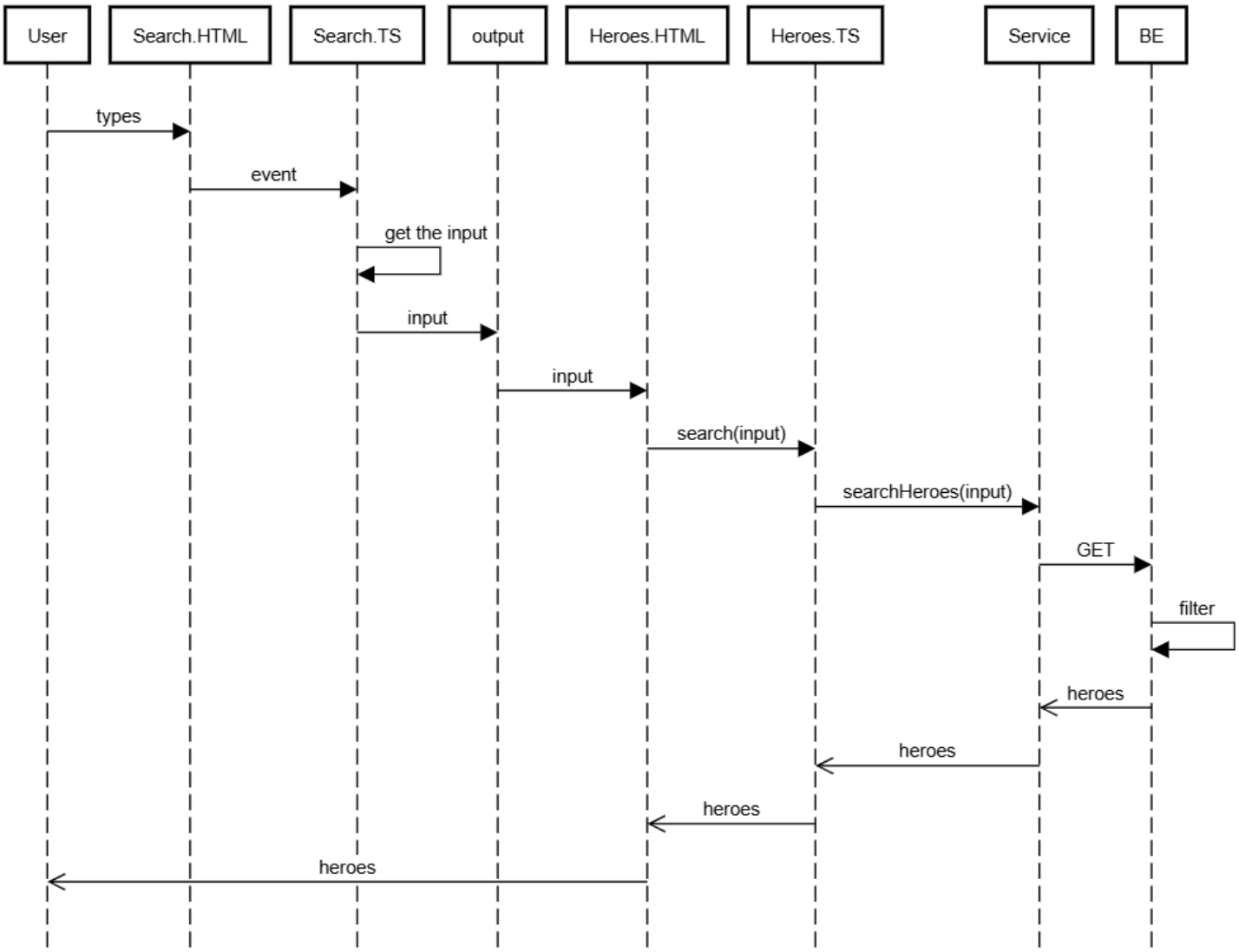
40. Finally, add the following style to `src/app/hero-search/hero-search.css`

```
#search-box {
  margin-left: 0.3rem;
  margin-bottom: 1rem;
}
```

41. Every time a user presses a button while the input field is in focus, an event is created. It contains a type of event (e.g., key is pressed), and a value (what symbol is added to the input field, for example). Also, Angular triggers the `onInput()` function in the class and passes the created event as a parameter. In the function, we take the value and send it to the `output` which is listened to by a parent component. Thus, we need to place the search component into a component that will be able to do something with the current value of the search field.
42. Add HeroSearch to imports of `src/app/heroes/heroes.ts` (will host searching) - `import { HeroService } from '../hero.service';` and `imports: [ HeroSearch, RouterLink ]`
43. Add the method to `src/app/heroes/heroes.ts`

```
search(text: string): void {
  text = text.trim();
  this.heroService.searchHeroes(text)
    .subscribe(heroes => this.heroes.set(heroes));
}
```

44. And, finally, in `src/app/heroes/heroes.html`, replace `<app-hero-search></app-hero-search>` with `<app-hero-search (searchChange)="search($event)"></app-hero-search>` - it will catch events that are sent to output by `HeroSearch` component and trigger the `search()` function in `Heroes` component for each of them.
45. Thus, the flow looks like
    a. A user types a symbol in the search field in the search component
    b. Angular catches the event and triggers `onInput()` function in the search component class passing the event as a parameter
    c. The function extracts the value from the event and sends it to the output `searchChange`
    d. The heroes component (that hosts the search component) listening for that output in `<app-hero-search (searchChange)="search($event)"></app-hero-search>` catches it and triggers the `search()` function in the heroes component class, passing the input there.
    e. The function calls the hero service passing the input.
    f. The hero service executes an API call to the backend with the input as a parameter and receives a filtered list of heroes.
    g. The function `search()` in the heroes component class receives the filtered list via an observable and puts that list into the `heroes` variable.
    h. As the `heroes` variable is a signal, it automatically triggers UI updates showing the filtered list of heroes to the user.



46. Switch to your browser and check that everything works as intended.

Improvements - Cut the requests

1. The search works, but every time a user presses a button, a request is sent to the backend, even if it's just the first symbol out of 5 that the user intends to print. Highly ineffective, especially with a slow connection.
2. Currently, as soon as a user presses a button, our search component immediately passes that information to the heroes component. We need to delay it and accumulate the changes so they would be sent only when the user stops printing. Additionally, the user can press buttons that are not symbols (e.g., left arrow) - it will trigger useless requests as well, and we would like to avoid it.
3. Add `import { debounceTime, distinctUntilChanged, Subject } from 'rxjs';` to `src/app/hero-search/hero-search.ts`
4. Add a variable `private searchText = new Subject<string>();` to `src/app/hero-search/hero-search.ts` - a `Subject` is an extension of an `Observable` that is provided by RxJS (a reactive library that does approximately the same as signals in Angular, but many would argue that it does it in a less intuitive way. Hence, signals in Angular were created)
5. Add the following method to `src/app/hero-search/hero-search.ts`

```
ngOnInit() {
  this.searchText.pipe(
    debounceTime(300),
    distinctUntilChanged()
  ).subscribe(value => {
    this.searchChange.emit(value);
  });
}
```

6. `pipe()` is a sequence of operations that are executed on every update to `searchText` one by one. `debounceTime()` is a delay in milliseconds that is introduced for each update before proceeding, and it is resetted every time `searchText` is updated, so if a user types “h” and then “e” before 300ms after “h” have passed, “h” won’t trigger a request and the timer will be resetted to 300ms, so a user can continue typing. `distinctUntilChanges()` compare the value after 300ms after the last update passed and the value before those updates started, so if a user typed “hello” and then deleted it immediately, no requests will be sent. It’s all possible due to `searchText` being a `Subject` that is essentially an `Observable` with some pepper, but it also means that we need to subscribe on the results of that `Observable` - when you are done with your operations, execute this code.
7. Finally, we need to route our event flow through `searchText`. To do it, update the method in `src/app/hero-search/hero-search.ts`

```
onInput(event: Event) {
  const input = event.target as HTMLInputElement;
  this.searchText.next(input.value);
}
```

8. Switch to your browser and check that everything works as intended.

Improvements - Error handling

1. Everything works in an ideal situation, but what if there is something wrong with the network connection or our backend is down? Our frontend application will silently do nothing. Let’s introduce some error handling.
2. Replace the method `getHeroes()` in `src/app/dashboard/dashboard.ts` with the following code

```
getHeroes(): void {
  this.heroService.getHeroes()
    .subscribe({
      next: heroes => this.heroes.set(heroes.slice(1, 5)),
      error: err => {
        console.error('Error fetching heroes:', err);
        alert('Failed to fetch heroes. Please try again later.');
      }
    });
}
```

3. Shutdown the backend, and try to open a dashboard in your browser - an alert should appear.
4. Note, that in our case we just show an error to a user in the most primitive way possible. In real systems, we can show custom popups, retry requests, fallback to alternative data sources or alter our application behaviour in any other required way.
5. Go through `HeroDetail` and `Heroes` components and add error handling to appropriate methods in a way you see fitting.
6. Note that we do not add error handling to the `HeroSearch` component as all it does is get an input from a user and pass it further.