# Angular

# Creating Angular project

- Install Angular globally
  - `npm install -g @angular/cli`

- Create new project
  - `ng new project-name`
  - `cd project-name`
  - `npm install`
  - `npm start  / ng serve`

# Adding Bootstrap

- Install with npm
  - npm install bootstrap

- Add to style in angular.json (it is in the root of project folder)

```
...
"styles": [
            "src/styles.scss",
            "node_modules/bootstrap/dist/css/bootstrap.min.css"
          ],
….
```

# Creating a New Component

- Each element manually… or using an ng command:
  - ng generate component component_name
  - ng g c component_name

- Add routing rule for the component
  - Import component at app-routing.module.ts

```
import { Page1Component } from './page1/page1.component';



const routes: Routes = [
  {path: 'page1', component: Page1Component}
];
```

# Capturing input in a component

- Import FormsModule
- This is the template driven way. Angular also has a “Reactive Forms” way.
- Forms in Angular are robust, but also can get a bit complex
- https://angular.io/guide/forms

```
import { FormsModule } from '@angular/forms';
[...]
@NgModule({
  imports: [
    [...],
    FormsModule
  ],
  [...]
})
```

# Capturing input in a component

- Html input component mapped to typescript class attribute
- [(ngModel)]: two-way binding (property + event)

```
<input class="my-2 form-control" id="item" type="text" placeholder="item" [(ngModel)]='item'>
```

```
.....
export class GroceriesInputComponent implements OnInit {

  item:String ='';
.....
```

# Capturing input in a component

- FormControls need attributes to have names, such as html inputs and selects
- Add [ngModelOptions]="{standalone: true}

```
<input class="my-2 form-control" id="item" type="text" placeholder="item" [(ngModel)]='item' [ngModelOptions]="{standalone: true}>
```

Or

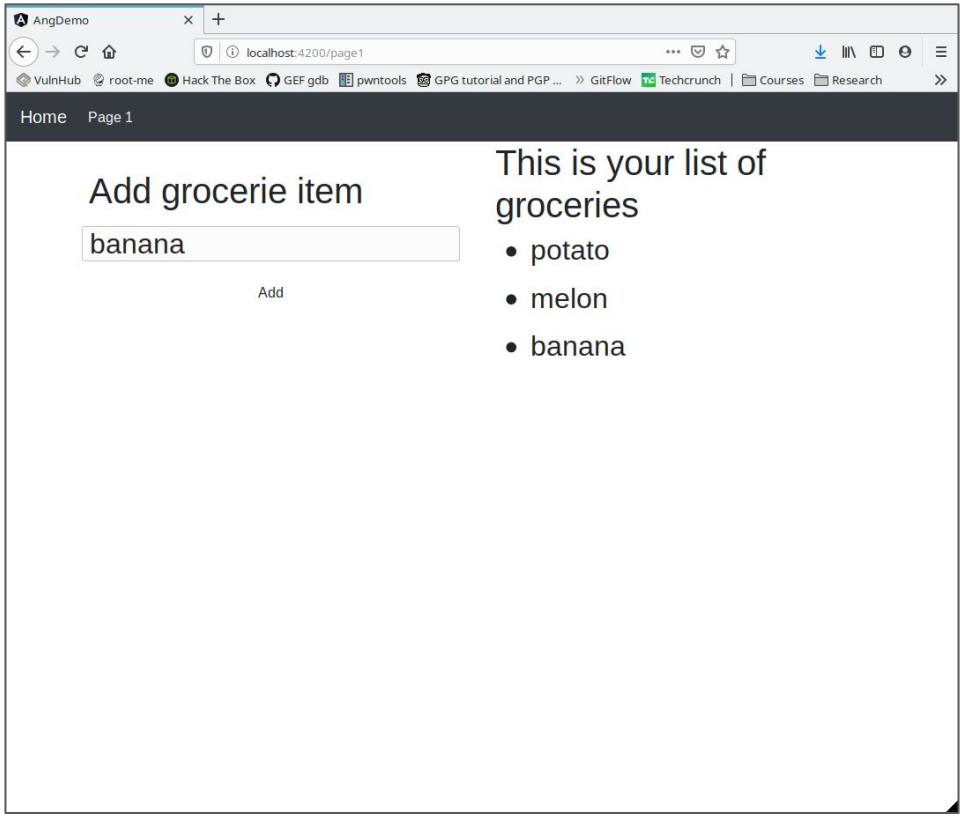
- Add name to input

```
<input class="my-2 form-control" id="item" type="text" placeholder="item" [(ngModel)]='item' name='inputItem'>
```

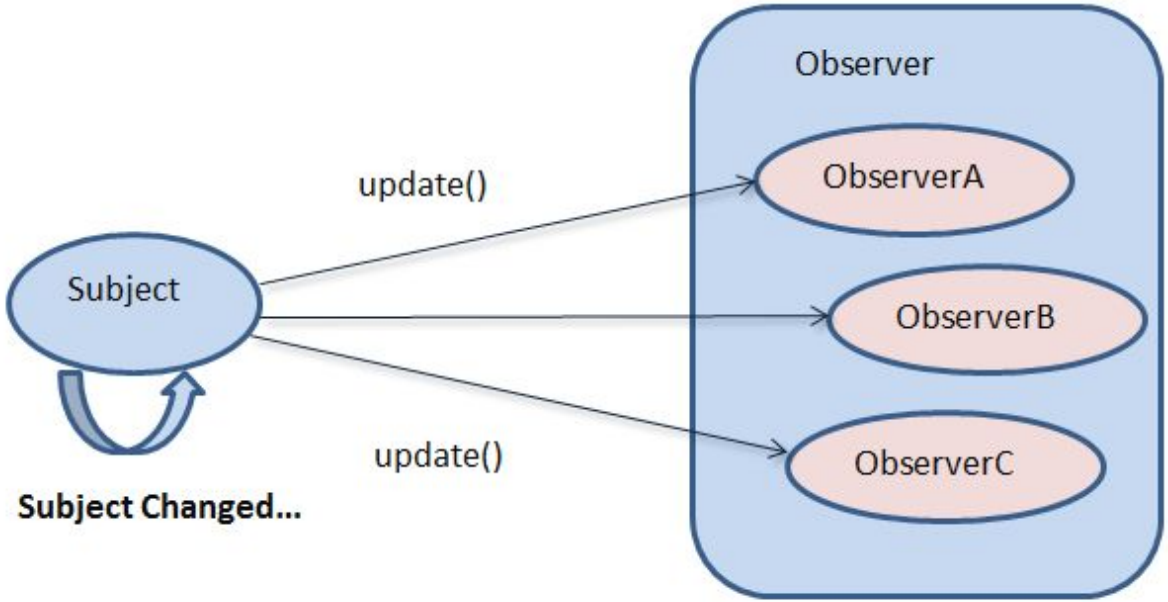
# Service

- ng command
  - ng generate service service_name
- Serves as a source of “truth” for data
- Injectable into components
- Can be declared with component scope (hierarchical) or application scope

# Live Coding Demo



# Observer Pattern



# EventEmitter

- An observable pattern
- Subscribed methods know when an event is triggered
- Used in services to signal data change

```
...
import { EventEmitter } from '@angular/core';

@Injectable()
export class GroceriesService {
  groceriesChanged = new EventEmitter<String[]>();
....
this.groceriesChanged.emit(changed data...);
...
```

# Live Demo steps

1. Create project (with routing)
2. Install Bootstrap
3. Create component page1
4. Create route for page1
5. Create component navbar
6. Create component groceries-input
7. Create component list
8. Make page1 show input and list components side by side (bootstrap grid)
9. Make input component capture an input list item with a click binding and a method
10. Make list show a list of strings (the grocories)
11. Create a service to store a list of groceries

# Live Demo steps

12. Inject service in list component

13. In ngOnInit() of list component make this.list receive service’s list

14. In input component, when adding the item, call service addItem, that adds the passed item to its string array.

15. To demonstrate this, add an item, go back to homepage clicking the navbar name (we should change it to home) and then back to “page1”, the list should have updated the items. If you refresh the page, the list will lose the item because it isn’t persisted and the memory is cleaned.

# Live Demo steps

16. To show data being dynamically updated, we need to use eventemitter in the service and subscribe to it in the list, as if using an observable.