Plan

1. Introduction
2. Setup of an Angular Project
3. Components
4. Expressions
5. Event Handling
6. Data Binding
7. Directives
8. Event Handling
9. Observables
10. Services
11. Services
Why Angular?

- http://www.madewithangular.com/
- cross-platform development
- template-based approach
- highly scalable platform
- good tool support
- continually improved and supported by Google
Angular Design Features

- modules: code encapsulated in separate modules/files
- directives: re-usable, modular functions and UI components
- services: re-usable tasks
- data binding: data and display of it are connected bi-directionally
- dependency injection: easy linking of dependencies
Installing Angular

prerequisites:

- node.js
- npm

run npm install -g @angular/cli
Installing Angular

prerequisites:

- node.js
- npm

run npm install -g @angular/cli

Exercise

Install angular on your machine.
angular allows to create an example starter project

- testing whether the framework and dependencies are working
- using an example project as a starting point for own project
Angular Example Project

ng new myproject

cd myproject

ng serve

then open http://localhost:4200/ in browser
Angular Example Project

```bash
g new myproject
cd myproject
ng serve

then open http://localhost:4200/ in browser
```

Exercise

Create an example angular project, open it in a browser, and inspect the structure of the generated directory.
3 top-level folders

- *e2e*: end-to-end testing info
- *node_modules*: dependencies, listed by *package.json*
- *src*: application source code, primarily in *app* directory
Angular Project Structure

(selected) files on the top level

- `angular.json`: angular configuration
- `package.json`: project name and dependencies
- `tsconfig.json`: TypeScript compiler options
- `tslint.json`: rules applied during compilation
Angular consists of modular building blocks
building blocks of different nature serve different purposes
goals:
  - step-by-step explore different buildings blocks
  - combine them to build applications
Components

components:

- composite widgets consisting of
  - a HTML template
  - a TypeScript code
- most basic, re-usable UI elements
- components often contain other components
Components

Component TypeScript files consist of 2 parts:

- **decorator section**: configuration
- **class section**: logic
Components

- in the Angular starter project: `app.component.ts` example component
- four files:
  - `app.component.ts`
  - `app.component.html`
  - `app.components.css`
  - `app.components.spec.ts`
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'angular7-app';
}
Components – Decorator Section

- decorator section: after imports, before class declaration
- in the Component declaration imported from angular core
- specifies a selector (part of HTML document this should be applied to)
- specifies operations associated with this part
Components – Decorator Section

- **selector**: name of (custom) element where this component is added
- **template**: inline HTML to add at selector location
- **templateUrl**: HTML file to add at selector location
- **styles**: inline CSS rules for selector location
- **styleUrls**: CSS rule file for selector location
Components – Class Section

- class is exported at the end of a component declaration
- typically: define a constructor to initialize values of the class (assign default values)
- concretely: set values for instance variables in the exported class
import { Component } from '@angular/core';

@Component({
  selector: 'simple-constructor',
  template: '
    <p>Hello today is {{today}}!</p>
  ,
})
export class UsingAConstructor {
  today: Date;
  constructor() {
    this.today = new Date();
  }
}

source: Dayley et al. [2017]
Generating and Linking Components

- In `src/app` run
  ```bash
  ng g package/component
  ```
- Automatically updates declarations in module declaration (src/app/app.module.ts), but necessary to
  - add the new component in bootstrap parameter
  - add the element the new component’s selector refers to HTML
- Components need to be in the declarations array of the module to be usable by other components
Adding a selector

- selector in component code
- framework automatically replaces HTML elements that are matched by selectors by the components defining the selector

⇒ in order to display a component, add a HTML element fitting the selector
Components

Exercise

Write a component that contains a start page for a web application. It should consist of a heading, a description text, and a “start” button. Adapt your module declaration so that instead of the default component, your new component is displayed when launching your angular app.
Expressions

- powerful feature of angular: data binding
- HTML/DOM and TypeScript/JavaScript code are connected
- instance fields from TypeScript can be referenced in the HTML code
- no need to write post-hoc code to insert or keep data and display synchronous
• represent data from components in HTML
• instance fields in classes in components are linked to expressions by name (updating one updates the other)
• syntax: double curly brackets
  `{{myExpression}}`
• expressions are not interpreted literally, but the result of evaluating them is displayed
Expressions

- certain operations supported inside expressions
- **Examples:**

```javascript
// formatting arrays
{{mynumbers.join("", "")}}

// adding values
{{score+5}}

// type and equality
{{a===b}}
```
Expressions – Pipes

- formatting of values of expressions handled by pipes
- examples: currency formatting, uppercase transformation, date formatting
- pipes can be chained together
- possibility to define custom pipes

https://angular.io/guide/pipes#pipes
Expressions – Example

excerpt from …component.ts:

})
export class ExprdemoComponent {

today:Date;

creator() {
    this.today = new Date();
}

updateToday() {
    this.today = new Date();
}
}
Expressions – Example

excerpt from ...component.html:

Today is {{today | date:’yyyy-MM-dd HH:mm:ss Z’}}

<a (click)="updateToday()">Update time stamp</a>
Event Handling

- Event handling: react to events triggered by user actions
- syntax: `<element (event)="function()">`
- possibility to pass `$event` to the function to pass event-specific data to the handler
- wide range of built-in events supported by angular
Event Handling

```html
<input (click)="displayEventData($event)"
(mouseenter)="displayEventData($event)"
 dblclick="displayEventData($event)"
keypress="displayEventData($event)"/>
...
displayEventData(event:any) {
    if(event.type == 'keypress'){
        this.eventcontent = "pressed " + event.key;
    }    
    
}
...  ```
Exercise

Define a new component for implementing expressions. Write an expression that displays the contents of a number array in a comma-separated string. Implement a button that is associated with a click handler that when clicked adds a random number to the number array.

Test your component: when the variable is updated in the click handler, the expression should automatically display the updated value.
properties of HTML elements can be bound to variables in classes

binding to TypeScript code instance fields allows to programmatically modify properties

syntax: `[properties]="aName"` with `aName` being an instance field in the component class

typical use case: change images on `src` elements
Data Binding – Property Binding – Example

html:

```html
<button [disabled]="buttonEnabled">A button</button>
<button (click)="switchButtonEnabled()">Another button</button>
```

ts:

```typescript
export class ExprdemoComponent {
    buttonEnabled: boolean;
    constructor() {
        this.buttonEnabled = true;
    }
    switchButtonEnabled() {
        this.buttonEnabled = !this.buttonEnabled;
    }
}
```
Data Binding – Class Binding

- class binding as a special case of property binding
- possibility to dynamically assign, change, or remove classes of HTML elements
- similar procedure of class binding applicable to style binding for inline CSS rules
Data Binding – Class Binding

html:

```html
<div [class.toclick]="classIsTrue">ClassIsTrue</div>
<div [class]="someClass">SomeClass</div>
```

ts:

```typescript
export class ExprdemoComponent implements OnInit {

    classIsTrue: boolean;
    someClass: string;

    constructor() {
        this.classIsTrue = true;
        this.someClass = "myclass";
    }

```
Two-Way Binding

- two-way binding to bind the value of an input field to a variable
- comparable to expressions, but allowing for user input
- possibility to programmatically access the value a user inputs into a field, or change it and display the result to the user
How to implement two-way binding:

1. Import into your module: `import FormsModule from '@angular/forms';`

2. Add this import into the `imports` parameter in the `@NgModule` decorator

3. Create a HTML element with `[(ngModel)]`:

   `<input [(ngModel)]="textVariable">`

4. Create a variable `textVariable` in the exported component class

The value of the textVariable is now bi-directionally connected between the code and HTML.
Structural Directives

- dynamically display or hide certain parts of HTML
- since HTML can be bound to data: display certain parts of data
- put logic into base HTML to dynamically render a page for each user
- dynamically add, remove, manipulate elements from the DOM
Structural Directives

- applying a structural directive to an element applies to its descendants, too
- different from hiding them via CSS: removed elements are completely removed from HTML and memory
- declared with an asterisk * and then the directive: *ngIf, *ngFor, *ngSwitch, ...
- directive added to element declaration, will be delivered dynamically to the client
Structural Directives – Example

Example:

```html
<div *ngIf="someFunction()">
Enter your name: <input>
</div>
```

The `div` and its content will only be shown if `someFunction()` evaluates to true.
Structural Directive – Example

<input type="checkbox" [(ngModel)]="nmode"> Night Mode<br>
<div class="nmd" *ngIf="nmode">Welcome to Night mode!</div>
Structural Directives

- `*ngFor` to define a loop over a collection of elements
- add a HTML template to every object in the referenced collection
- “microsyntax” to define the iteration
### Structural Directives – Example

**ts:**

```typescript
export class DirctivdemoComponent {
    brands:string[];
    constructor() {
        this.brands = ['GoodBrand', 'NiceProducts', 'HighQuality', 'PerformanceBrand', 'FashionExpert'];
    }
}
```

**HTML:**

```html
<div *ngFor="let brand of brands; let i=index">  
    <div class="card">{{i + 1}}: {{brand}}</div> 
</div>
```
Angular supports attribute directives to modify attributes of elements
useful application: read out form values
Example – Reading out Form Values

<form #loginForm="ngForm"
(ngSubmit)="onSubmit(loginForm.value)">
  Name: <input type="text" name="username" ngModel>
  Password: <input type="password" name="userpassword" ngModel>
  <button type="submit">Submit</button>
</form>
Note the `ngModel` statement at the end of the input elements. The value of the `name` attribute is translated into a variable and can be accessed programmatically:

```javascript
onSubmit(formContent) {
  let uname = formContent.username;
  let upass = formContent.userpassword;
  console.log(uname, upass);
}
```
Exercise

Implement a component that makes use of **property binding**, **class binding**, **two-way binding** and **structural directives** to implement a night mode. The component should contain

- a checkbox to toggle the night mode
- an image placed in the assets directory
- a heading or a text above the image
- a structural directive with some text

Toggling the checkbox should replace the image with a night mode-optimized image, change the background and font color of the text, and make the text in the structural directive visible.

http://localhost:4200/props
Custom Directives

- define custom behavior of HTML elements
- reusable TypeScript code attachable to DOM
- similar to components, but with a difference:
  - components are re-usable UI components with logic
  - directives are re-usable logic for existing UI elements
steps for implementing custom directives:

1. create a Directive decorator with a selector attribute
2. the selector attribute defines an attribute this directive applies to
3. add a special attribute to every element this directive should be applied to
4. export a class where the logic of the directive is defined
Event Handling: Built-In

built-in events:

- add event in brackets to element declaration
- assign a function to the value of the event in brackets
- example events: (click), (change), (focus), (submit), (keyup), (keydown), (keypress), (mouseover)

```
<input type="text" (change)="myEventHandler($event)" />
```
Event Handling: Custom Events

- possibility to define own events when specific conditions are met
- add event emit code to a component
- register custom event handlers to catch the emitted events
- main usage:
  - transfer data between components
  - communicate from child to parent component
Event Handling: Custom Events

- embed a component within other components
- task: react in the parent component to events in the child component
- pass data out of the child element by sending custom events up the component hierarchy
Event Handling: Custom Events – Example

child component HTML:

```html
<button (click)="customFunction()">Click</button>
```
child component TypeScript:

```typescript
@Component({
  selector: 'app-custevtdemo',
  ...
  @Output() customEvent: EventEmitter<any> = new EventEmitter();
  ...
  customFunction(){
    this.customEvent.emit(
      "text to be transferred");
    console.log("emitted custom event")
  }
  ...
})
```
Event Handling: Custom Events – Example

parent component HTML:

```html
<app-custevtdemo (customEvent)="handleCustomEvent($event)" >
</app-custevtdemo>

<h4>{{mytext}}</h4>
```
parent component TypeScript:

```typescript
handleCustomEvent(event:any){
    this.mytext = "caught custom event: " + event
}
```

...
Observables

- Observables allow to subscribe and unsubscribe to parts of the application where data changes

Examples:
- keystrokes
- HTTP responses
- timers
"Observables are lazy collections of multiple values over time."

https://medium.com/@luukgruijs/understanding-creating-and-subscribing-to-observables-in-angular-426dbf0b04a3
Observables

- ability to subscribe and unsubscribe to an observable with an observer
- comparable to a newsletter subscription:
  - unregular updates over time
  - lazy: newsletter only sent to subscribers
  - newsletter changes over time with new content
  - ability to unsubscribe at any time
Observables

- Observables push and don’t pull
- pull: observer decides when to fetch data
- push: data source decides when the observer receives data
import {Observable} from 'rxjs/Observable';

if the package can not be found:

npm install --save rxjs-compat
constructor() {
    let simpleObservable = new Observable((observer) => {
        observer.next("hey there listeners");
        observer.complete();
    });

    let mysubscr = simpleObservable.subscribe(
        data => {console.log(data); this.text =""+data;}
    );
    mysubscr.unsubscribe;
}
Observables

- angular uses Observables extensively ‘under the hood’
- more powerful than e.g. change handlers since it can provide push functionality for external resources
- possibility to make use of the functions and apply them explicitly
Services

- service: re-usable code portion
- often concise code portion that is re-used in multiple places
- integrated into other code portions via dependency injection
- pre-defined, built-in services like `http`
- angular offers to build custom services
Dependency Injection in Components

- dependency injection to add code at run time from one file into another
- write code only once and reuse it in other classes by dependency injection
- typically services injected into classes
dependencies injected via *providers*

- providers declared in decorators
- different parameters for different decorators
  - in `@Component` of `@NgModule`, use `providers = []`
  - in the service declared with `@Injectable`, use the `providedIn = ''` parameter to associate the service with a provider, typically 'root'
Built-In Services

Examples of built-in services:

- **http**: send & receive HTTP requests
- **router**: navigation functionality
- **forms**: input forms with validation
http Service

- supports different types of requests, e.g. GET or POST
- asynchronous callback as a sequential Observable
- allows to process data as it comes back in
http Service: How to import

in `app.module.ts`:

```typescript
import { HttpClientModule } from '@angular/common/http';
```

in `mycomponent.ts` or `myservice.ts`:

```typescript
import { HttpClient } from '@angular/common/http';
```
import { HttpClient } from '@angular/common/http;

private http: HttpClient;

this.http.get("/myUrl");
this.http.post("/myUrl", data);

this.http({method: 'GET', url:'/myUrl'});
http Service – Callbacks

- Observable\<Response>\n- Observable provides pre-defined methods that can be used to every part of the response
- alternative: 'classical' Observable subscription

```typescript
this.http.get("/myUrl")
.subscribe((data: Config) => {this.mydata = data;});
```
alternative way: cast to a promise

promises wait until the data is complete (\texttt{then(...)}\texttt{)} in contrast to Observables (lazy collection of values over time)

JSON as a data structure that is natively compatible (return values from server as JSON)
http Service – Callbacks

```javascript
http.get('./assets/dummyDB.JSON')
  .toPromise()
  .then((data) => {
    this.users = data.JSON()
  })
  .catch((err) =>{
    console.log(err);
  })
```
http Service

- recommended by angular docs: define the HTTP request as a service and inject when needed
- handle errors and data postprocessing there
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
    constructor(private http: HttpClient) { }

    showConfigResponse() {
        this.configService.getConfigResponse()
            // resp is of type 'HttpResponse<Config>'
            .subscribe(resp => {
                // display its headers
                const keys = resp.headers.keys();
                this.headers = keys.map(key =>
                    '${key}: ${resp.headers.get(key)}');

                // access the body directly, which is typed as 'Config'.
                this.config = { ... resp.body }
            });
    }
}

source: https://angular.io/guide/http
router service

- routing: different parts of the application available via different URL patterns
- alternative way of displaying **components**: instead of HTML selectors, map components to URL patterns
- routes are stored in the browser history allowing to navigate back and forth
router service

- at project creation time: prompt asks whether to enable routing
- adds necessary files and imports and allows to start directly with the actual contents
router service

- routing logic should be defined in a routing module
  `src/app/app-routing.module.ts`
- import this module in `app.module.ts`:

```typescript
import { AppRoutingModule } from './app-routing.module'
```

```typescript
@NgModule(
  imports: [
    BrowserModule,
    AppRoutingModule,
    ...
  ]
)...
```
import the components to be mapped, e.g.
import CustevtparentComponent from './custevnt/custevtparent/custevtparent.component';

define routes in a Routes object like
const routes: Routes = [
  path: 'observables', component: ObsdemoComponent,
  path: 'eventparent', component: CustComponent,
...]

define the module IOs in @NgModule decorator:

  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })
Routes URL mappings:

- order matters, first-match strategy
- more specific routes should be placed before less specific ones
- default path: empty path, when starting the application
- last path: wildcard ** to catch any URL pattern not mapped (unknown or mistyped patterns)

https://angular.io/guide/router
const routes: Routes = [
  { path: 'events', component: EvntdemoComponent },
  { path: 'start', component: PlaceholderComponent },
  { path: '', redirectTo: '/start', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponentComponent }
];
changing the URL from within the application:

- define `routerLink="/mypattern"` attribute to an element in a component’s HTML

Example:

```
<h2 routerLink="/start">Welcome to Angular!</h2>
<button routerLink="/eventparent">Click</button>
```
elements will be added as a sibling to 
<router-outlet></router-outlet>
when the URL changes, the sibling component of router-outlet will be exchanged dynamically
alternative: programmatic change of URL

Router singleton in Angular applications can be accessed

pass Singleton into constructor, then call it

calendar

constructor(private router: Router) {}  
...
this.router.navigate(['myRoute']);
Services

- re-usable functionality
- comparable to static functions in Java
- written once as injectable code parts
- injectable into components or modules
Examples of services:

- data transformations
- HTTP interfaces
- constant data service
- shared service for multiple components
Services – How to

step 1: add Injectable decorator

import { Injectable } from '@angular/core';

@Injectable()
export class CustomService { }

Björn Rudzewitz (University of Tübingen)
Services – How to

step 2: import in component and add to providers array

```typescript
import { CustomService } from './path_to_service';

@Component({
  selector: 'app-root',
  template: '',
  providers: [ CustomService ]
})
```

Björn Rudzewitz (University of Tübingen)

Angular

Tübingen, November 26, 2019 87 / 92
Services – How to

step 3: instantiate in component’s constructor

```typescript
// the constructor for creating the component
constructor(
  private myServ: CustomService
){}
```

step 4: use functions from service

```typescript
this.myServ.functionFromService()
```
angular command line interface allows to create services:

`ng generate service myservice`
Deploying

building the app: `ng build --watch`

`ng build --prod`
then put the generated folder on a server (alternative, advanced types of deployment exist)
Add a routing module to your application. Instead of placing several elements in your `index.html` file, add only one entry-point component to your application that contains elements with `router-link` tags interpreted by the application’s routing module.

official angular documentation: https://angular.io/docs