Angular

Challenges of Web client-side development

- Q: What will Project 3 demo look if we implement it as a single HTML + JS + CSS file?
- Code complexity
  - Something started as simple UI code gradually became full-blown desktop code
  - tens of thousands of lines of JavaScript code
- Lack of modularity
  - An app consists of many small components of different functionality
    - e.g., folder tree, file list, menu bar, ...
  - But they are all placed in a single HTML, CSS, JavaScript code
  - Also, the "code" for one component are spread across HTML, CSS, JavaScript files
  - Difficult to manage and reuse "codes"

Angular overview

- JavaScript (or TypeScript) framework for client-side development
- Supports development of complex single-page application (SPA)
  - Modules, reusable components, testing, etc
- An angular app is composed of a set of independent *components*
  - A component is a JavaScript class that is responsible for a part of the application
    - e.g., folder tree, file list, menu bar, ...
  - A component (class) is associated with an HTML *template* (and CSS style) that describes its presentation on the page
    - Model-View-Controller (MVC) or Model-View-ViewModel (MVVM) pattern
  - *Data binding* is used to "communicate" between the HTML template ("view") and its component class ("controller" or "view model")

    * interpolation, property binding, event binding, two-way binding

Angular CLI (Command-Line Interface)

Running Example: Google suggest in Angular (AppComponent + SearchBoxComponent + DisplayComponent)

- Angular comes with a set of tools to:
  1. Generate the initial skeleton code for an app

     ```
     $ ng new google-suggest
     ```

     - Most important codes are in `src/app`
     - Angular CLI creates the top-level "app component" and includes it in the "root module" (more on modules later

     ```
     $ ls src/app
     app.component.css    app.component.spec.ts   app.module.ts
     app.component.html   app.component.ts
     ```

     - `app.component.ts`, `app.component.html`, `app.component.css` are component (class), template and CSS file
     - `app.module.ts` is the root module of the app
  2. Dynamically compile, build, and serve the app through a simple HTTP server

     ```
     $ ng serve
     ```

  3. Build the final "app" that can be deployed to a simple HTTP server

     ```
     $ ng build
     ```

     - A set of .html, .css, .js files are produced at `dist/` directory
     - Once built, these files can be deployed to any Web server
     - In principle, nothing needs to run on the HTTP server. Everything runs on the client as a JavaScript program!
  4. Generate the skeleton code for component, service, module (`ng generate component/service/module my-name`)

     ```
     $ ng generate component search-box
     $ ng generate component display
     ```

```
$ ls src/app
app.component.css      app.component.ts  search-box/
app.component.html     app.module.ts
app.component.spec.ts  display/
```

- kebab-case vs camelCase
  - Angular uses
    - camelCase in JavaScript code
    - kebab-case in html, filenames
  - This mixture is because
    - Many file systems and HTML are *not* case sensitive
    - JavaScript *is* case sensitive, but dash is not allowed in identifiers
    - We cannot consistently use kebab-case or camelCase everywhere!

Core concepts

- Angular applications are written by composing *HTML templates*, writing *component* classes to manage those templates, adding application logic in *services*, and boxing components and services in *modules*.
- *Component*
  - A specific part of the application responsible for certain UI
  - An application consists of multiple components
    - e.g., AppComponent, SearchBoxComponent, DisplayComponent
  - Each component is associated with an HTML template (and CSS style)
- *Template*
  - HTML with additional angular specific markup

```
// search-box.component.html
<div id="display">Suggestion here</div>

// display.component.html
<div><form action="http://www.google.com/search">
  <input type="text" name="q"><input type="submit">
</form></div>
```

  - *Directive*: "angular-specific markup"
    - *Component directive, structural directive, attribute directive*
- Q: How can I include SearchBox and Display components in the application?
  - *Component directive*

```
// app.component.html
<app-search-box></app-search-box>
<app-display></app-display>
```

    - Custom-defined "tag" that represents a component
  - Replace the content of `app.component.html` with above and show what happens

Component Decorator

- Q: How does the system know that `<app-search-box>` tag corresponds to `SearchBoxComponent`?
  - Through `@Component` decorator and its metadata

```
// display.component.ts
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-display',
  templateUrl: './display.component.html',
  styleUrls: ['./display.component.css']
})
export class DisplayComponent implements OnInit {
  constructor() { }
  ngOnInit() { }
}
```

  - `@Component` decorator also has info on its template and CSS files

Data Binding

- Q: How can a component and its template interact? Set input box value from property value? Call class method from input box?
  - *Data binding*: mechanism to exchange data between component class and template
    - *Interpolation*
    - Attribute directive: *Property binding*, *event binding*, *2-way binding*
- *Interpolation*
  - Q: `AppComponent` has `title` property. Can we display its value in its template?
  - Syntax: `{{ expression }}`
    - Replace `expression` with its output string
  - Example

```
// app.component.ts
title = "Google Suggest!";

// app.component.html
<h1>{{ title + " Application" }}</h1>
```

  - An identifier in `expression` must be either a *template variable* (more on this later) or a *property* of its component
  - Expression should not have any side effect
- *Property binding*
  - Q: Can we enable submit button only if input box is nonempty?
  - Syntax: `[property]="expression"`
    - Set the value of `property` to the result of `expression`

        - For an HTML element, `property` is either its *dom property* (or an *angular directive*)
        - For a component, `property` is the property of the component
    - Example

```
// search-box.component.html
<input type="text" name="q" #query><input type="submit" [
   disabled]="!query.value">
```

        - For an HTML element, property is the *DOM property* of the element
        - *template reference variable*
            - Syntax: `#varName`
            - A unique name given to an element, so that it can be referenced by others
    - Example

```
// app.component.html
<app-search-box [defaultQuery]="title"></app-display>
```

        - Set the value of `defaultQuery` of `SearchBoxComponent` to `title` of `AppComponent`
        - `@Input()` decorator
            - But the above change causes error!
                - A template can access *only the properties of its own component*
                - Template for `AppComponent` is trying to access a property of `SearchBoxComponent`!
            - Add `@Input()` decorator to allow other templates to access a property

```
// search-box.component.ts
import { Input } from '@angular/core';
...
@Input() defaultQuery: string;
...
```

    - Property binding can be used to send data from parent to child component
- *Event binding*

- Q: Can we show alert message when the user presses the submit button?

- Syntax: `(event)="statement"`
  - Execute `statement` when `event` is triggered
- Example

```
// search-box.component.html
<input type="submit" (click)="showAlert()">

// search-box.component.ts
showAlert() { alert("Submit button pressed!"); }
```

  - Alert window pops up when the button is pressed
  - Any identifier in `statement` must be either a property (or method) of the component or a template variable
  - `statement` may reference `$event`, the “event object”
    - For a DOM element event, `$event` is the native DOM event object (e.g., `$event.target.value`)
    - For custom event, `$event` is what is “emitted” by `EventEmitter`
    - More on this issue later
  - `statement` may have side effect

- *Two-way binding*
  - Q: Data flow in all examples so far are one way. Interpolation: component -> template, Property binding: component -> template, Event binding: template -> component. Can we make data flow both ways?

    - Two-way binding can be done with what we have learned, but it is cumbersome and tedious
  - Syntax: `[(ngModel)]="property"`
    - `[()]` symbol indicates two-way data flow

        * `ngModel` is a Angular directive for two-way binding
        * Data flows both ways between `property` and the input box value
    - Example

```
// search-box.component.html
<input type="text" name="q" [(ngModel)]="query">
```

```
// search-box.component.ts
export class SearchBoxComponent {
    ...
    query: string;
    ...
}
```

        * Data flows both ways between `query` and the input box
            ▸ But by default, `ngModel` directive is not available in Angular
        * To use `ngModel`, **import** `FormsModule` **in** `AppModule`:

```
// app.module.ts
import { FormsModule } from '@angular/forms';

@NgModule({
    imports: [
        FormsModule,
    ],
```

            ▸ Any module listed in "imports" of `AppModule` is made available everywhere in the app

Angular Module System

- *NgModule*
    - Angular's own modularity system
    - Every Angular app has at least one NgModule class, the *root module*, often named `AppModule`

  - A cohesive block of code dedicated to a specific application or a closely related set of capabilities
  - Created with the `@NgModule` decorator

- Example

```
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { SearchBoxComponent } from './search-box/search-box.
   component';
import { DisplayComponent } from './display/display.component'
   ;

@NgModule({
  declarations: [
    AppComponent,
    SearchBoxComponent,
    DisplayComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
  ],
  providers: [
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

  - `declarations`: the set of classes that belongs to the module
    * Differently from standard JavaScript modules, classes in the same module may be split across multiple files
    * By default, classes in a module are “local” and cannot be imported and used by other modules

  - `exports`: the classes in this module that other modules can import and use
  - `imports`: the modules whose exported classes are used in this module
    * Very similar to `import` statement in JavaScript module
  - `providers`: the services that will be auto-created and injected through dependency injection (more on this later)
  - `bootstrap`: The root component of the app. Only root module has this property

- Commonly used modules
  - `import { BrowserModule } from '@angular/platform-browser`: When the app runs in a browser
  - `import { FormsModule } from '@angular/forms`: When we use form specific directives, such as `ngModel`
  - `import { RouterModule } from '@angular/router`: When we use routing, such as `routerLink`
  - `import { HttpClientModule } from '@angular/common/http`: When we use `HttpClient`

Summary So Far

- We have learned:
  - How we create a component and make it as a child of another component
  - How to exchange information between a template and a component through data binding
  - Angular module system
- Our next goal: Let us implement dynamic suggestion functionality
- Q: What do we need to do to provide dynamical query suggestions from Google?
  1. Monitor user input in `SearchBoxComponent`
  2. Send user input to the google suggest server
  3. Display response from google server in `DisplayComponent`
- Monitor `input` event in the input box and bind it to a method in `SearchBoxComponent`

```
<input type="text" name="q" (input)="sendQuery($event.target.
   value)">
```

  - Inside event binding, `$event` points to the DOM `event` object of the event
  - `$event.target` points to the DOM element to which the event was fired
  - `sendQuery()` must send the query to Google server, get suggestions, and display it in `DisplayComponent`

```
// search-box.component.ts
http: XMLHttpRequest = new XMLHttpRequest();

sendQuery(query: string) {
  this.http.open("GET", "http://google.com/complete/search?
     output=toolbar&q="+encodeURI(query));
  this.http.onreadystatechange = (() => this.processSuggestion
     ());
  this.http.send();
}

processSuggestion() {
  if (this.http.readyState != 4) return;

  let result = [];
  let s = this.http.responseXML.getElementsByTagName('
     suggestion');
  for (let i = 0; i < s.length; i++) {
    result.push(s[i].getAttribute("data"));
  }
  // pass suggestions to DisplayComponent
}
```

  - Note: Because of same-origin policy, the code works only if CORS is enabled on the server
    * Install and use "CORS extension" on chrome to get around this issue during development

- Q: How can a method in `SearchBoxComponent` send data to `DisplayComponent`?
  - Note: A template/component can access their own properties and method, but not others'

- Two popular ways to exchange data between sibling components
  - custom event generation + template reference variable
  - services

EventEmitter for Inter-component Communication

- Main idea for Approach 1
  - `SearchBoxComponent` cannot directly interact with its sibling, but it can "emit" an event
  - Parent template can "bind" to the event and pass information to `DisplayComponent` using *template reference variable*
- Implementation
  1. `SearchBoxComponent` emits `suggestion` event when it receives response from server
     - Pass suggestions as the event object
  2. `AppComponent` binds to `suggestion` event of `SearchBoxComponent`, and set `suggestions` property of `DisplayComponent` to the passed suggestions
  3. `DisplayComponent` displays `suggestions` in its template

  - Q: How can we throw `suggestion` event from `SearchBoxComponent`?
- `EventEmitter`
  - `EventEmitter` allows emitting a custom event from any component through `emit(event)` call
  - Example

    ```
    import { EventEmitter, Output } from '@angular/core';
    ...
    @Output() suggestion = new EventEmitter<string[]>();
    ...
    this.suggestion.emit(result);
    ```

    * `@Output()` decorator allows other components to bind to this event

        * The component triggers `suggestion` event and emits the event object `result`

- Q: Where can I catch `suggestion` event and pass it to `DisplayComponent`?
    - Bind to `suggestion` event of `SearchBoxComponent`

      ```
      // app.component.html
      <app-search-box (suggestion)="display.suggestions=$event">
         </app-search-box>
      <app-display #display></app-display>
      ```

        * `#display`: *template reference variable*
            - Syntax: `#varName`
            - A unique name given to an element, so that it can be referenced by others
    - Add `suggestions` property to `DisplayComponent`

      ```
      // display.component.ts
      import { Input } from '@angular/core';
        ...
      @Input() suggestions: string[];
      ```

    - Q: How can we display the array of suggestions in the display template? For loop inside template?

      ```
      // display.component.html
      <ul>
          <li>{{ suggestions??? }}</li>
      </ul>
      ```

- *Structural directive*: `*ngIf`, `*ngFor`, `*ngSwitch`

  ```
  // display.component.html
  <ul>
      <li *ngFor="let suggestion of suggestions">{{ suggestion
         }}</li>
  </ul>
  ```

  - `*ngFor="let a of A"` creates one DOM element per each item in the array `A`
    - `a` is a *template input variable*. A variable created inside a template, not component
    - In case of name conflict, template variables has precedence to component properties
  - `*ngIf="expression"` adds the element and its descendants to the DOM only if the expression is true (= not falsy)

Service and Dependency Injection

- Main idea for Approach 2
  - Use a third-party “messenger” to exchange data between independent components!
  - `service` in Angular
- In Angular, service is an independent JavaScript class that
  1. implements complex application logic or
  2. works as a communication channel between components

  - Implementing complex application logic
    - Methods of each component are used mainly for view animation and simple user interaction
    - Anything more complex than simple user interaction is implemented in a separate service
  - Mechanisms for exchanging information between components
    - Service
    - Parent-child property binding (parent -> child)
    - Parent-child event binding (child -> parent)
    - Event binding and template reference variable (sibling <-> sibling)
    - See https://angular.io/guide/component-interaction for detail
- Creating a service

```
$ ng generate service suggestion
$ ls -l
suggestion.service.spec.ts      suggestion.service.ts
```

- What `SuggestionService` should provides:
  - `sendQuery(query)`: Let any component send a query to Google server
  - `subscribe(callback)`: Let any component register callback for response from Google

```
// suggestion.service.ts
callback = null;
http = new XMLHttpRequest();

subscribe(callback)
{
  this.callback = callback;
}

sendQuery(query: string) {
  this.http.open("GET", "http://google.com/complete/search?
     output=toolbar&q="+encodeURI(query));
  this.http.onreadystatechange = (() => this.processSuggestion
     ());
  this.http.send();
}

processSuggestion() {
  if (this.http.readyState != 4) return;

  let result = [];
  let s = this.http.responseXML.getElementsByTagName('
     suggestion');
  for (let i = 0; i < s.length; i++) {
    result.push(s[i].getAttribute("data"));
  }
  if (this.callback) this.callback(result);
}
```

  - `SearchBoxComponent` will call `sendInput(query)` whenever user input is detected
  - `DisplayComponent` registers its callback function, so that it will be called when

suggestions arrive
  - **Give students time to digest and understand the code**

- Q: Who needs access to `SuggestionService`? Which component uses the service?
- Q: Who should "create" `SuggestionService`? `SearchBoxComponent`? `DisplayComponent`?
  - A service typically does not "belong to" any particular component
  - It is a shared "service" among many components
  - The main application itself, not individual components, should create a service and make it available to everyone
- **Dependency Injection**
  1. The service that needs to be created at the application level is listed in `providers` attribute of the `AppModule`
  2. Any component that needs to use the service list it as a parameter of its constructor
  3. When the application starts, the `AppModule` creates an instance of the service class and passes the created instance as a constructor parameter

```
// app.module.component
import { SuggestionService } from './suggestion.service';
...
providers: [ SuggestionService ],
```

  - `AppModule` automatically creates any service listed in `providers` and pass it to any component who need it.

```
// search-box.component.ts
import { SuggestionService } from '../suggestion.service';
...
constructor(private suggestionService: SuggestionService) { }

// display.component.ts
import { SuggestionService } from '../suggestion.service';
...
suggestions: string[];
```

```
constructor(private suggestionService: SuggestionService) {
    suggestionService.subscribe(suggestions => this.
       suggestions = suggestions);
}
```

  - Any class that needs a service just have to add it as a constructor parameter

- A few more minor changes

```
// search-box.component.html
<input type="text" name="q" (input)="suggestionService.
   sendQuery($event.target.value)">
```

  - Change `search-box.component.html` to use the service instead of its own method

```
// app.component.html
<app-search-box></app-search-box>
<app-display></app-display>
```

  - Remove template reference variable and custom event handling

Other Topics

- **Router**
  - Angular's `RouterModule` helps dealing with the browser back button and supporting "deep links"
    - Provide URL to component mapping
    - Allows bind to "URL activation" events
  - Read Routing & navigation section of Angular documentation to learn more detail
- **Forms**
  - Manipulating and interacting with forms is a common tasks of most Angular apps
  - Angular provides a number of different ways to support this interaction
    - Template-drive forms, reactive forms, . . .
  - Read Forms section of Angular documentation to learn more detail

- **Pipes**
  - To "transform" data for output in template can be done using *pipes* `|`
    - Makes it easy to format data inside a template
  - Read Pipes section of Angular documentation to learn more detail

Summary of Core Angular Concepts

- Component
- Template
- Directive
  - Component directive
  - Attribute directive
  - Structural directive
- Data binding
  - Interpolation
  - Property binding
  - Event binding
  - Two-way binding
- Template variable
  - Template reference variable
  - Template input variable
- Service
- Dependency injection
- NgModule
- Inter-component communication
  - EventEmitter
  - Input, output decorator
- Routing

References

- Angular tutorial: https://angular.io/tutorial
- More extensive book on Angular (free): https://codecraft.tv/courses/angular/
- Official Angular documentation: https://angular.io/guide/architecture