IN4MATX 285: Interactive Technology Studio

Programming: Interface Frameworks

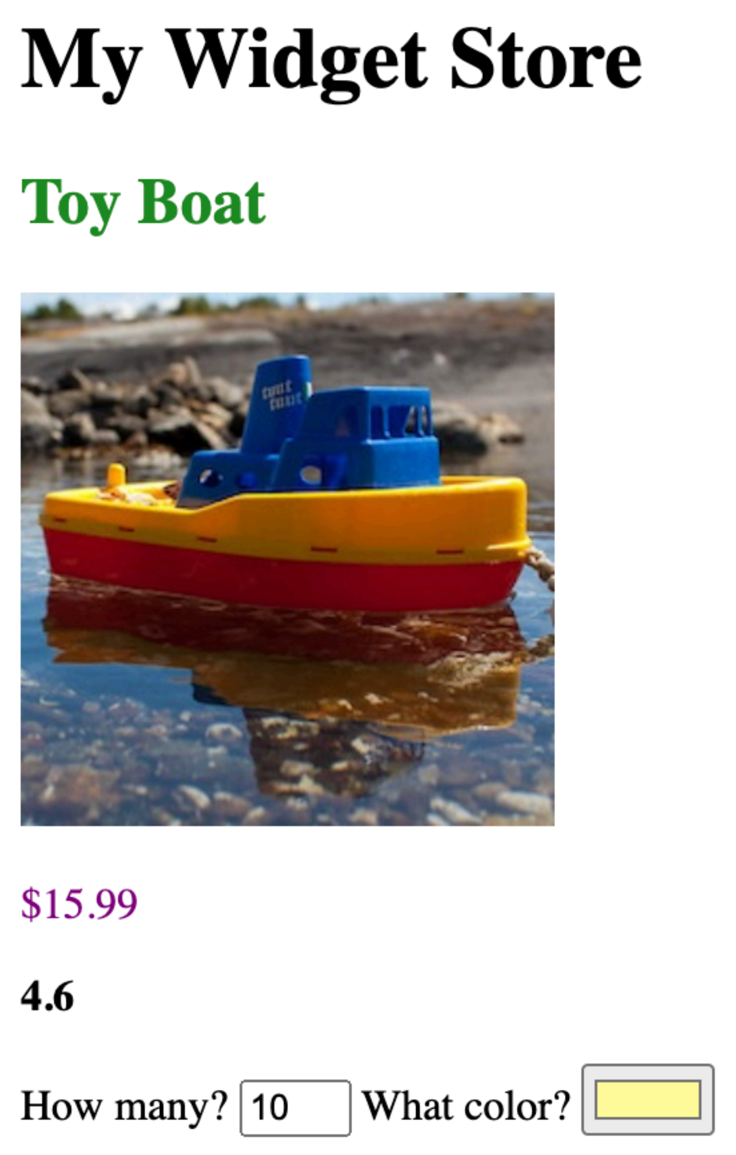
Today’s goals

By the end of today, you should be able to…

- Describe the objective of frameworks toward developing complex interfaces
- Explain a Model-View-Controller Architecture and how Angular implements the architecture
- Describe the role of an Angular component
- Broadly navigate Angular’s file structure

A “small” client interface

- 3 pages
- Limited interactivity between pages; each page was fairly self-contained
- Interface was static, not personalized to an individual user



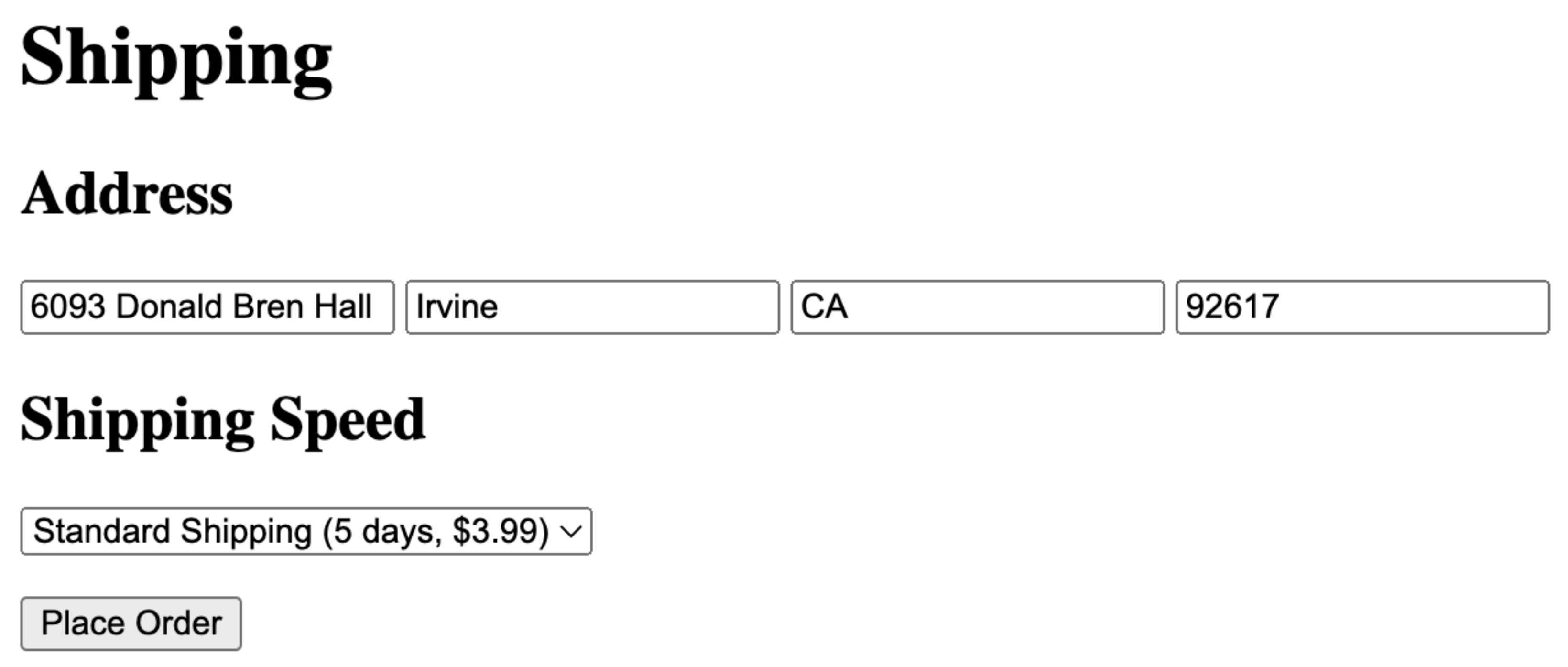
My Widget Store

Toy Boat

$15.99

4.6

How many? 10 What color?



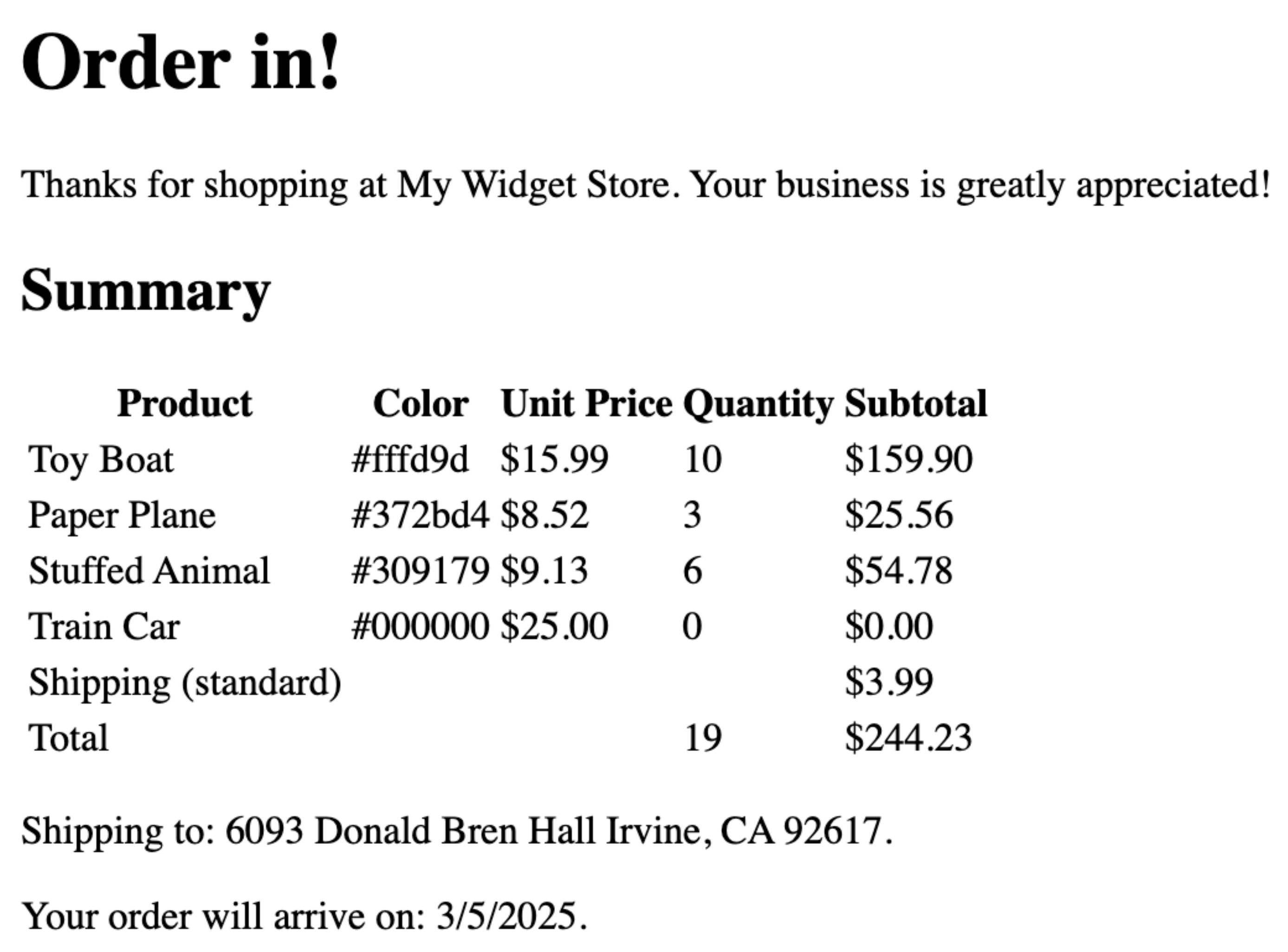
Shipping

Address

6093 Donald Bren Hall | Irvine | CA | 92617

Shipping Speed

Standard Shipping (5 days, $3.99)



Order in!

Thanks for shopping at My Widget Store. Your business is greatly appreciated!

Summary

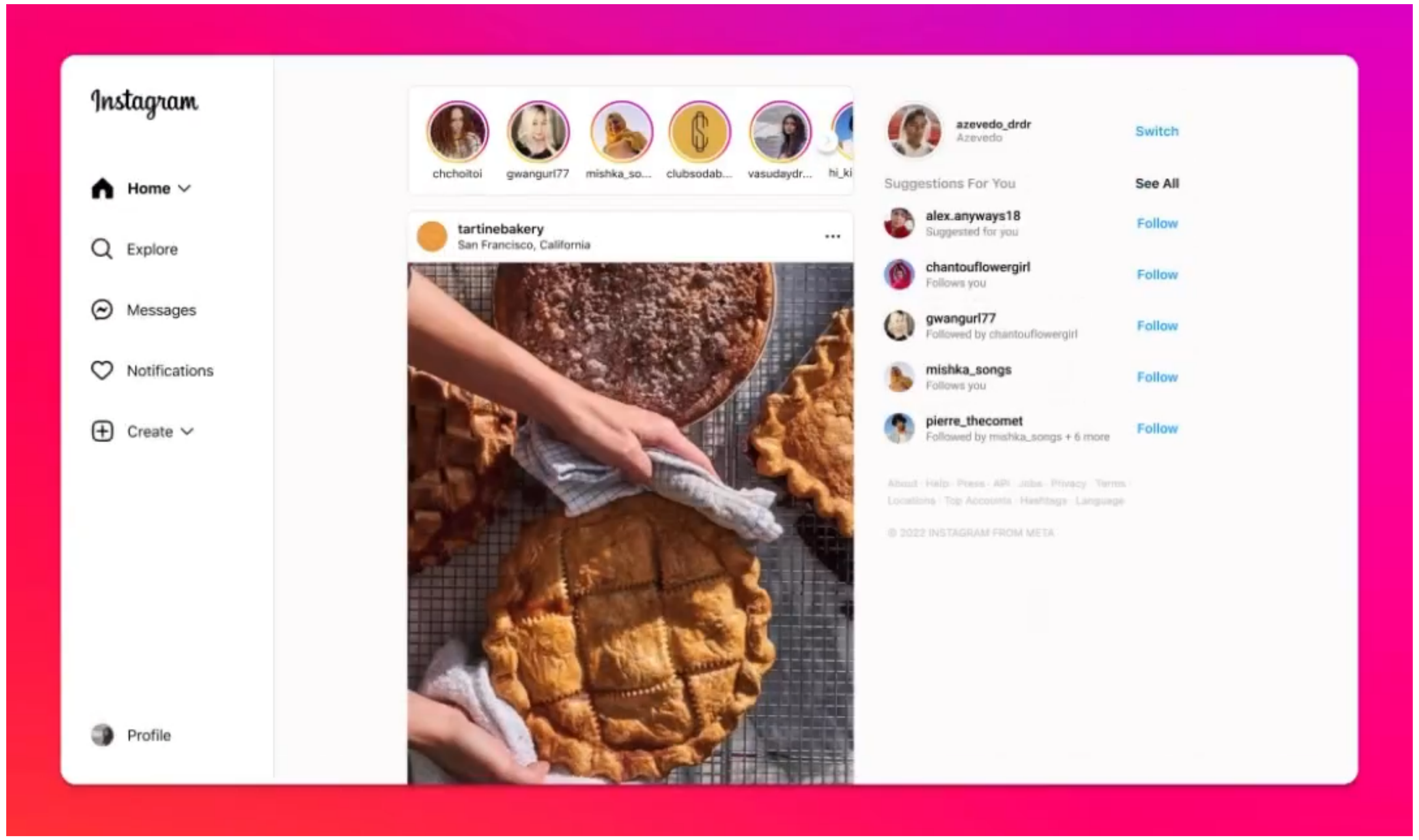
Product	Color	Unit Price	Quantity	Subtotal
Toy Boat	#fffd9d	$15.99	10	$159.90
Paper Plane	#372bd4	$8.52	3	$25.56
Stuffed Animal	#309179	$9.13	6	$54.78
Train Car	#000000	$25.00	0	$0.00
Shipping (standard)				$3.99
Total			19	$244.23

Shipping to: 6093 Donald Bren Hall Irvine, CA 92617.

Your order will arrive on: 3/5/2025.

A “large” client interface

- Hundreds of pages and ways to navigate between pages
- Repeated UI components (posts, heart button)
- Different content, links, etc. displayed for each person



How do developers create large client applications?

Frameworks for large clients

- Add structure and organization
- Make UI components reusable
- Support modularity
  - Import packages, UIs, etc. when needed

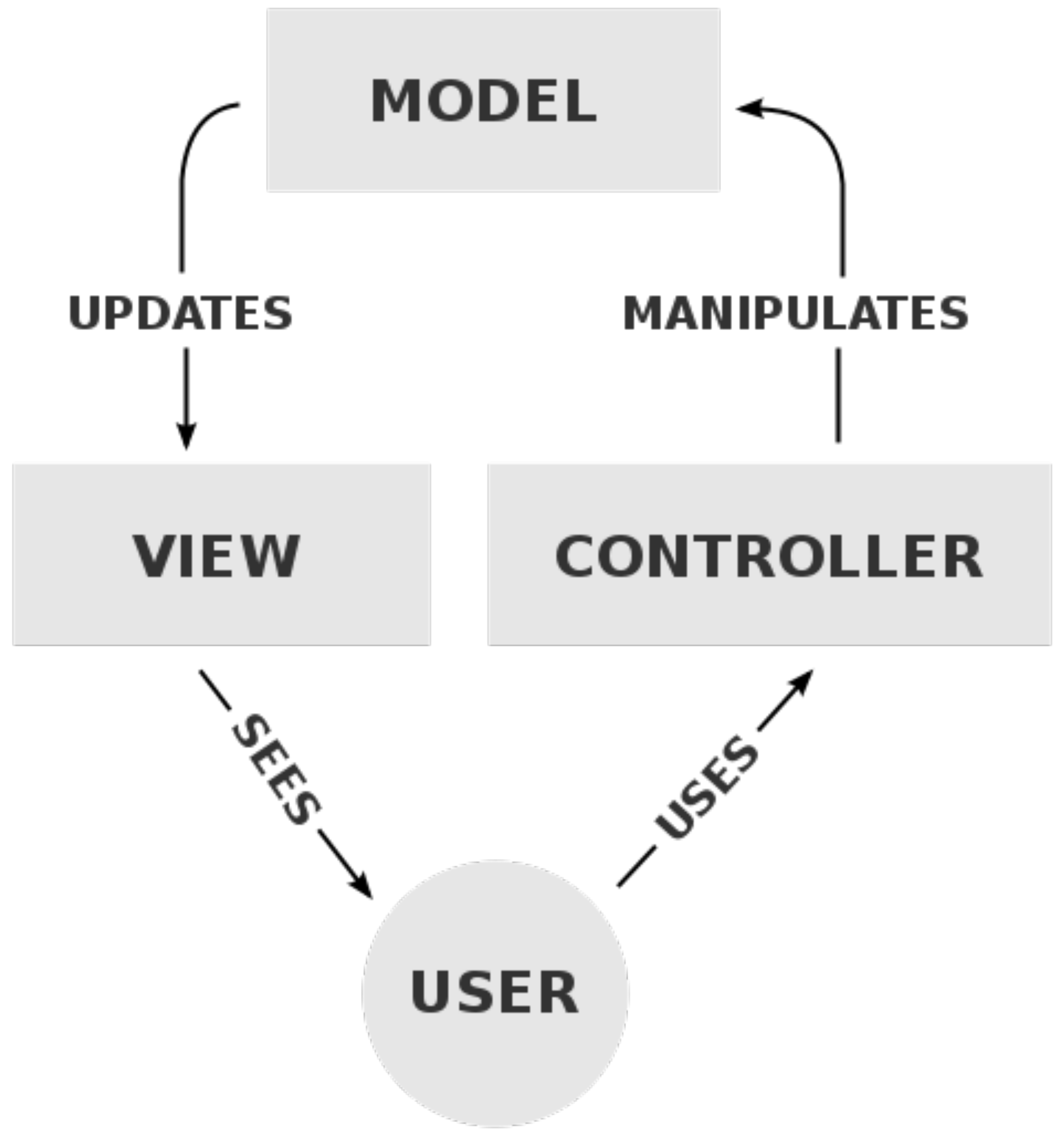
Frameworks for large clients

- Angular
- React
- Vue.js
- (Insert your favorite other framework here)
- All support the same overall goal
- All have commonalities in how they function

Angular architecture

Model-View-Controller

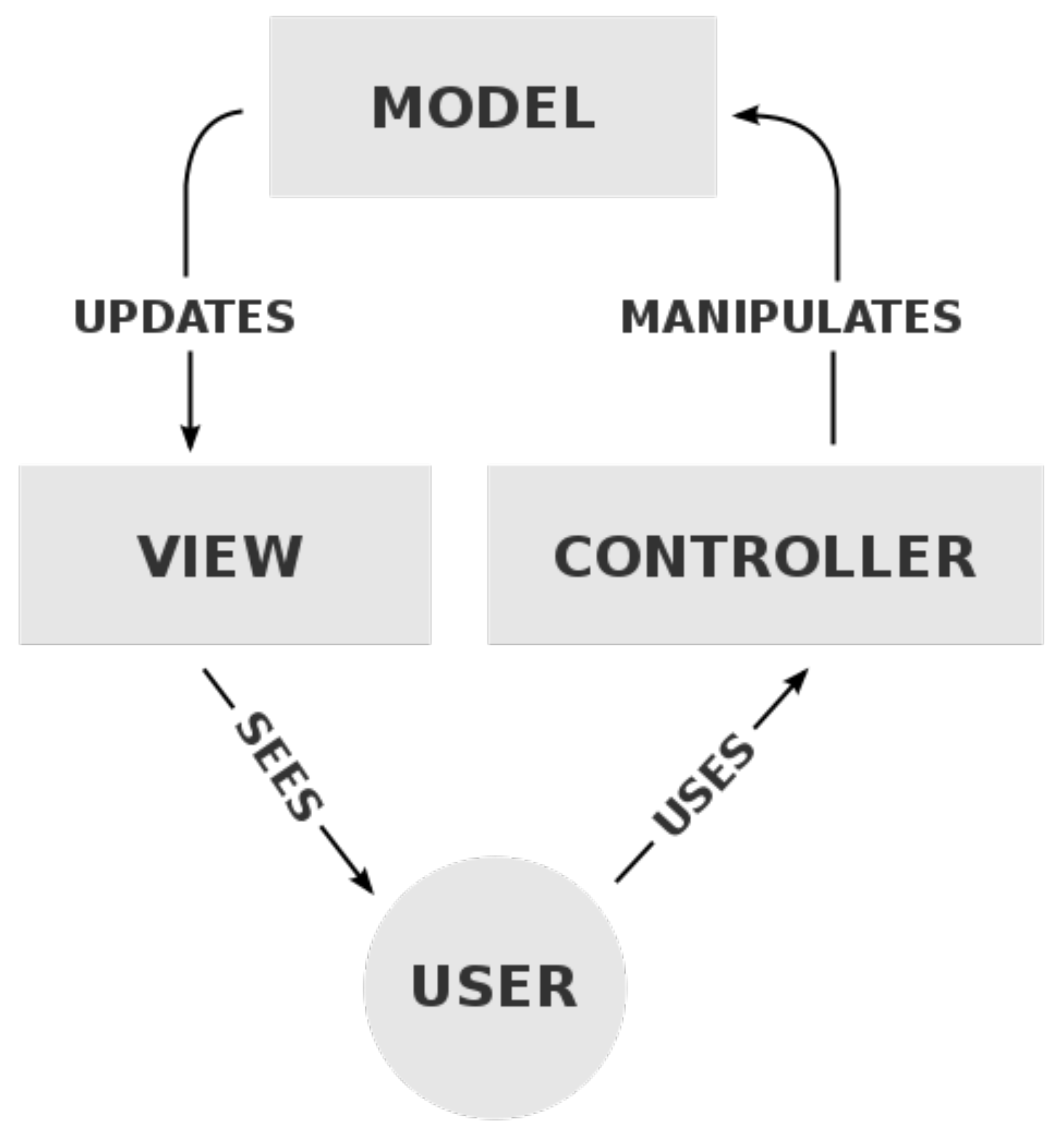
- Approach for structuring the code behind interfaces
- Model: the data behind an app
- View: the visual interface of an app
- Controller: the interaction with an app



https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Model-View-Controller

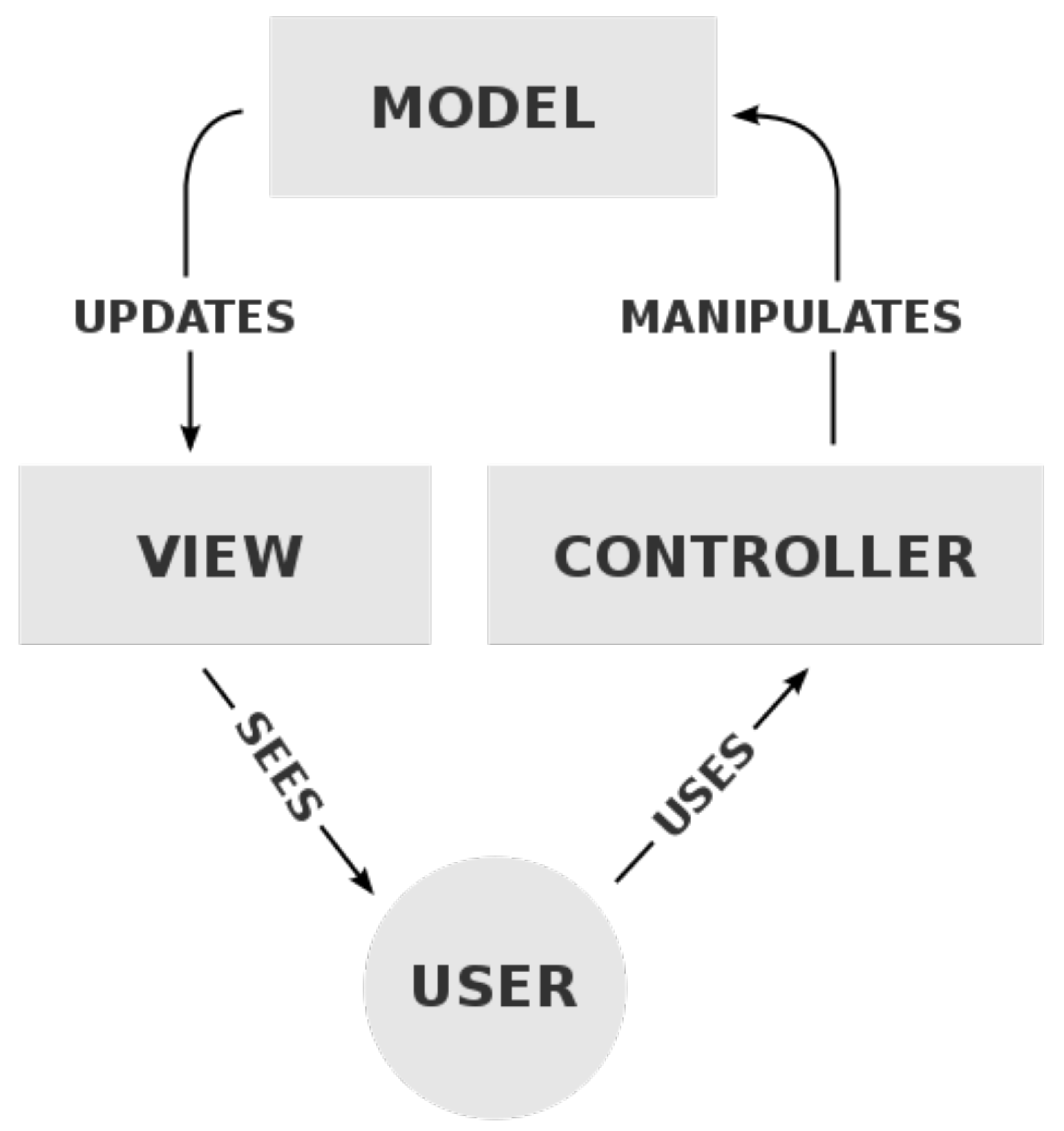
- Model: the data behind an app
  - Notifies views when it changes
  - Enables views to query the model for data
  - Allows the controller to manipulate data in the model



https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Model-View-Controller

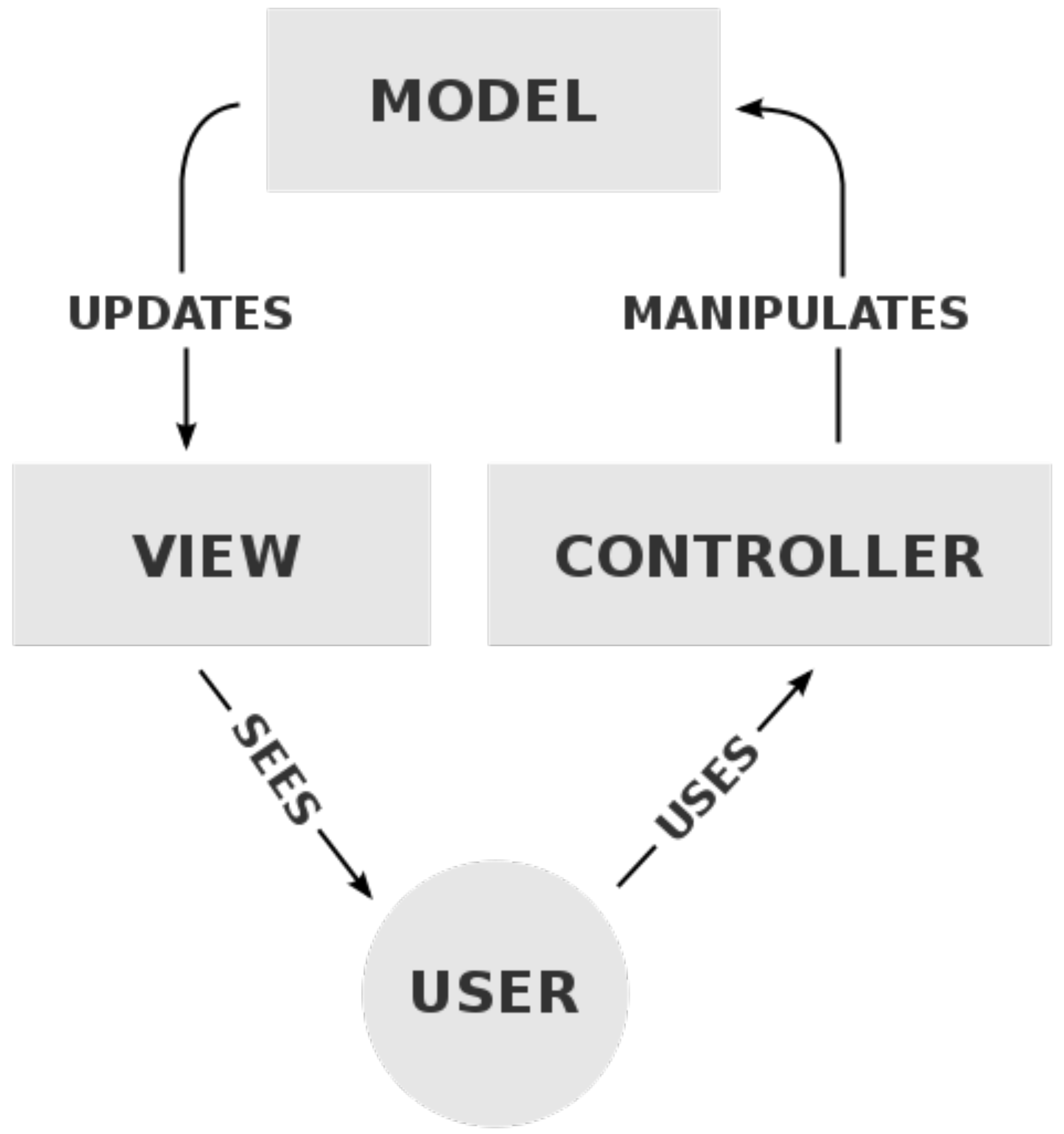
- View: the visual interface of an app
    - Renders the contents of the model
    - Specifies how the model data should be presented
    - When the model changes, the view
      must update it's presentation
    - "Push" approach: the view waits for change notifications
      (live updating feed)
    - "Pull" approach: the view must ask when it wants new data (pull to refresh)
    - Forwards input to the controller



https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Model-View-Controller

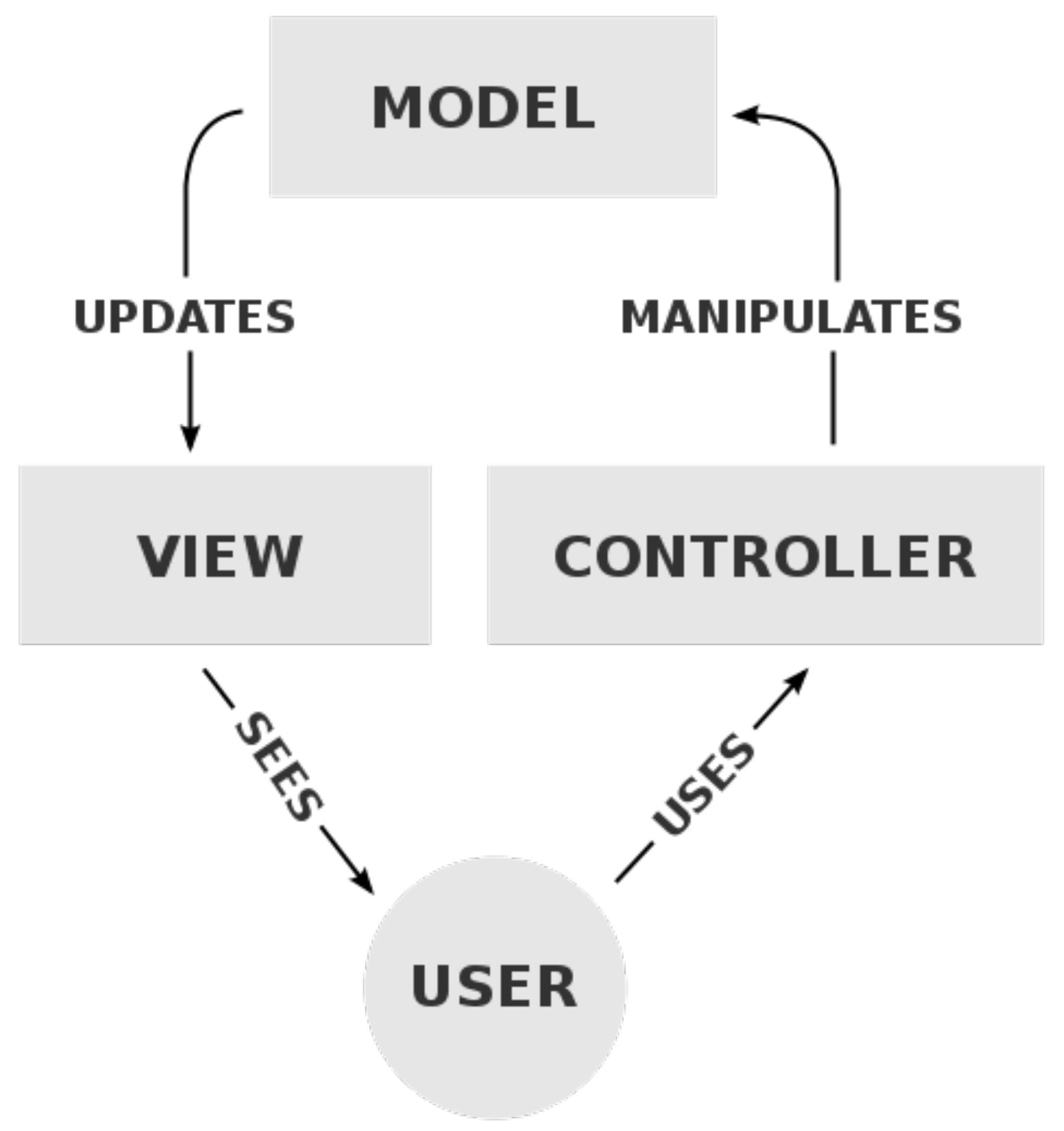
- Controller: the interaction with an app
  - Interprets user input and maps them to actions
  - Tells the model what actions to perform
  - Indirectly tells the view (through the model) if page should be rendered differently



https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Model-View-Controller

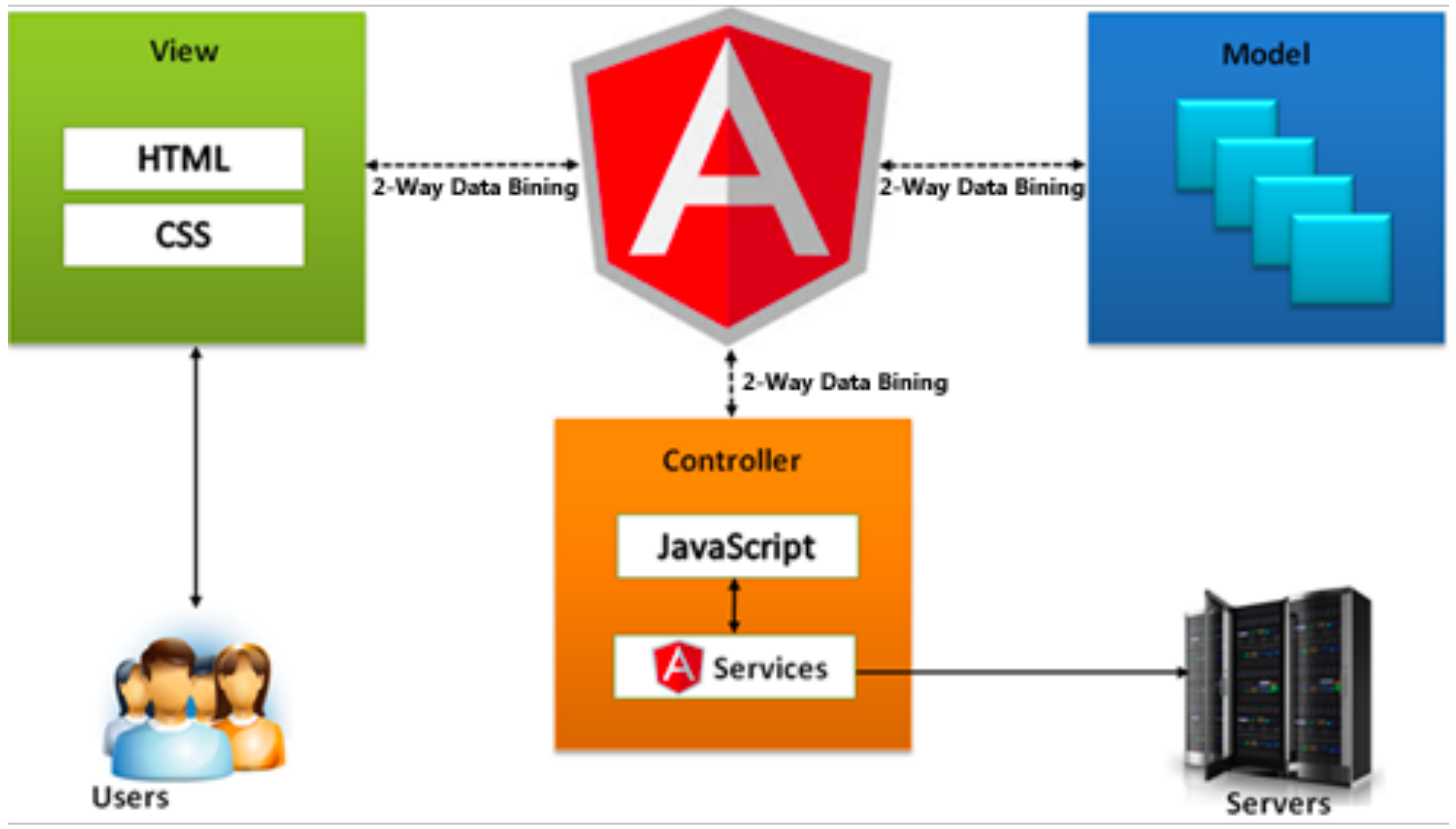
- Model: JavaScript for loading, parsing, and manipulating data
- View: HTML and CSS to specify layout
- Controller: event handlers for buttons and inputs in JavaScript



https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

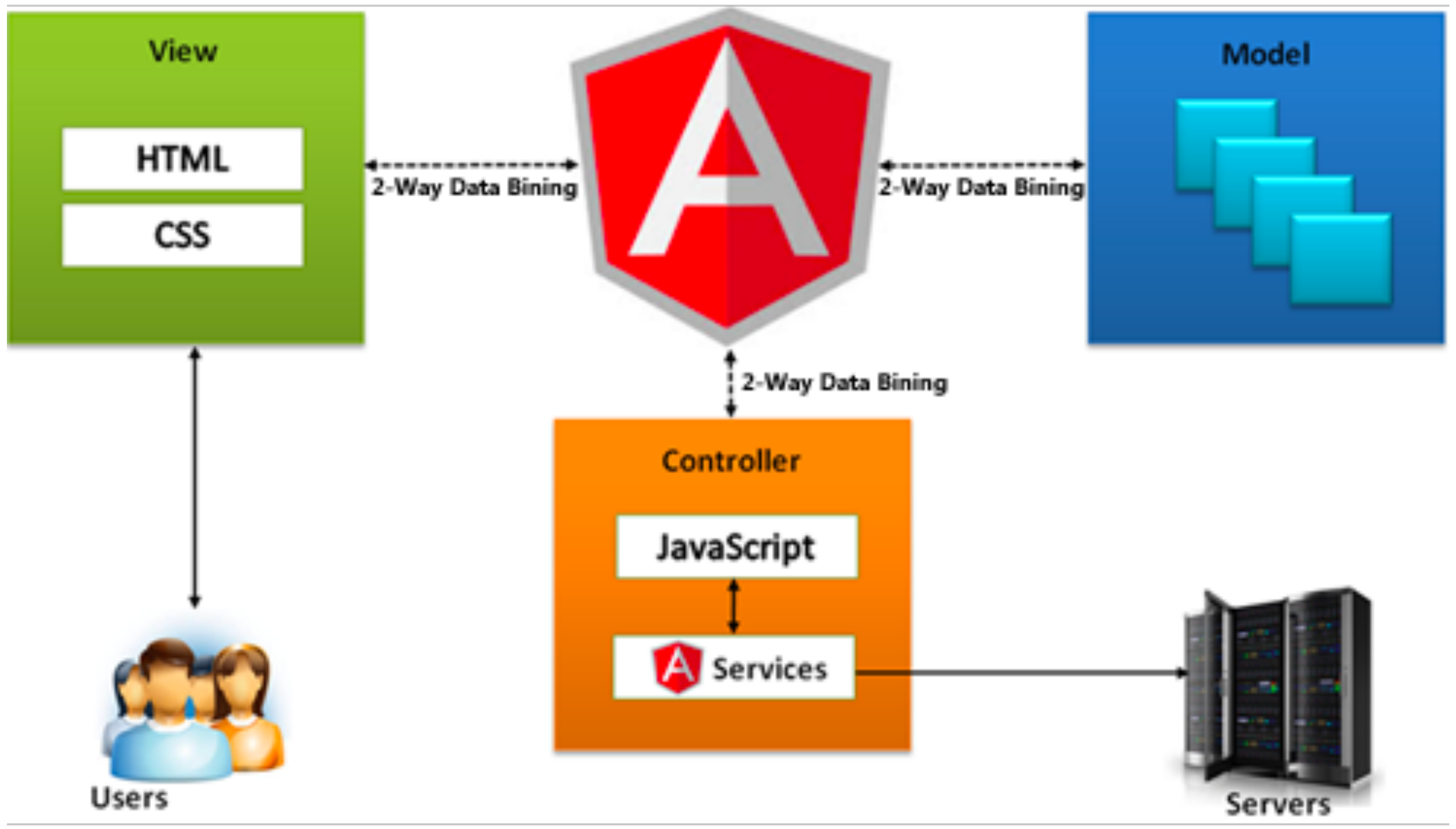
MVC in Angular

- View: HTML and CSS
- Model & Controller: JavaScript
- Angular functionality serves as the glue between the three



MVC in Angular

- **Binding**: key term
  - Variables in a view can be *bound* to variables and functions in a model or controller
  - When a variable in the *model* changes, any references to it in the *view* will also change ("push" model)
  - When a view receives input from a user, it passes it to the controller bound for that input



Following MVC in Angular

Angular components

- A component is an interface element
  - Usually larger than “a button”, but smaller than “a page”
  - Usually one which repeats across the interface

Angular components

- Defines the model, view, and controller for any interface element
- Each component makes a folder consisting of four files:
  - `hello.component.css` (view)
  - `hello.component.html` (view)
  - `hello.component.spec.ts` (for automated testing; we'll mostly ignore)
  - `hello.component.ts` (model and controller)

Binding in Angular

Four types of binding

- Interpolation: {{ }}
- Property: []
- Event: ( )
- Two-way: [( )]

https://angular.io/guide/template-syntax

Interpolation binding {{ }}

- “Weave calculated strings into the text between HTML element tags and within attribute assignments”

```
<h3>
  {{title}}
  <img src="{{heroImageUrl}}" style="height:30px">
</h3>
```

https://angular.io/guide/template-syntax

Property binding [ ]

- “Set an element property to a component property value”

```
<img [src]="heroImageUrl">
```

https://angular.io/guide/template-syntax

Event binding ( )

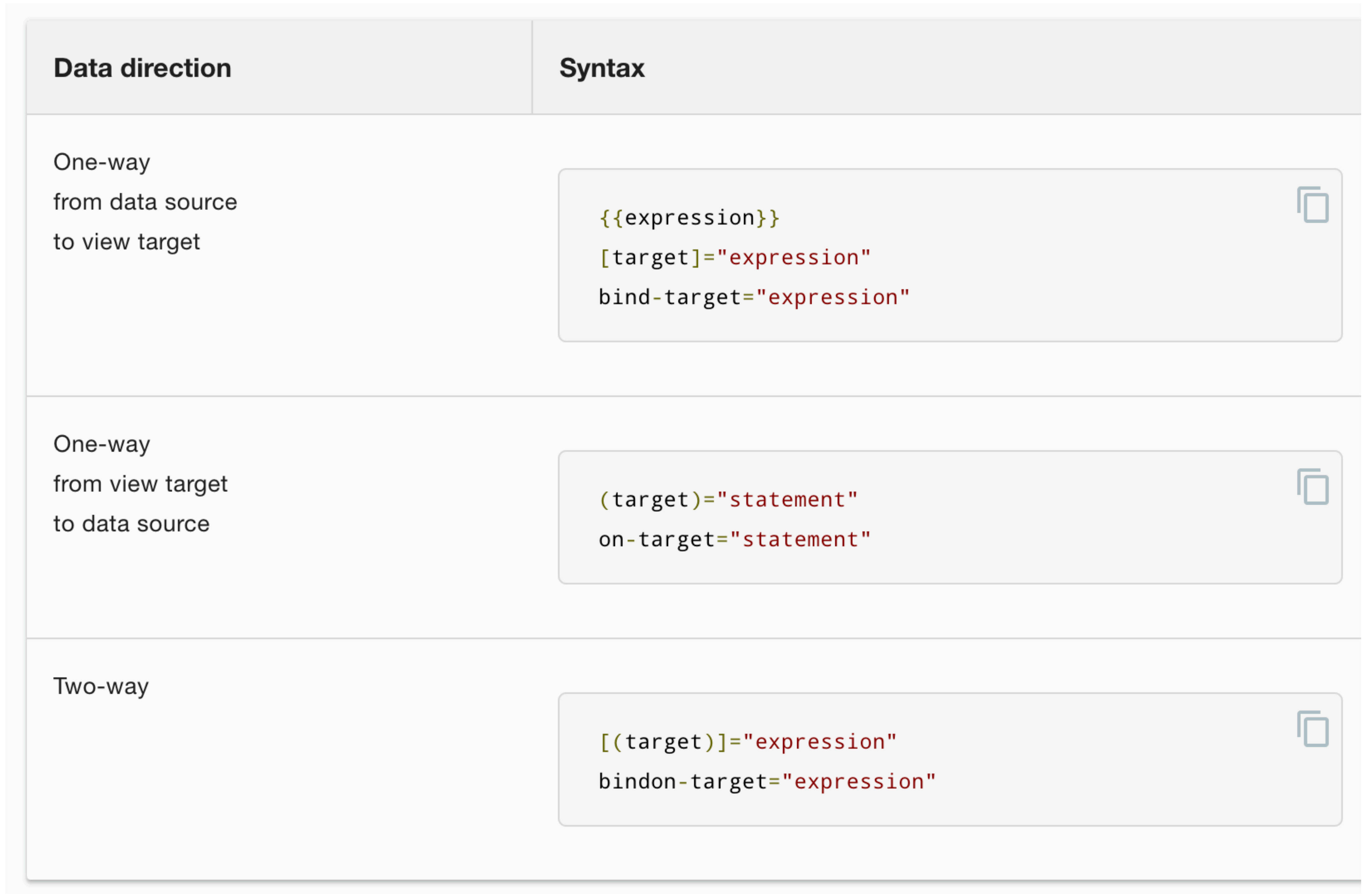
- “Listen for certain events such as keystrokes, mouse movements, clicks, and touches”

```
<!--When clicked, will run the onSave() function in
component.ts file-->
<button (click)=“onSave()">Save</button>
```

https://angular.io/guide/template-syntax

One-way binding

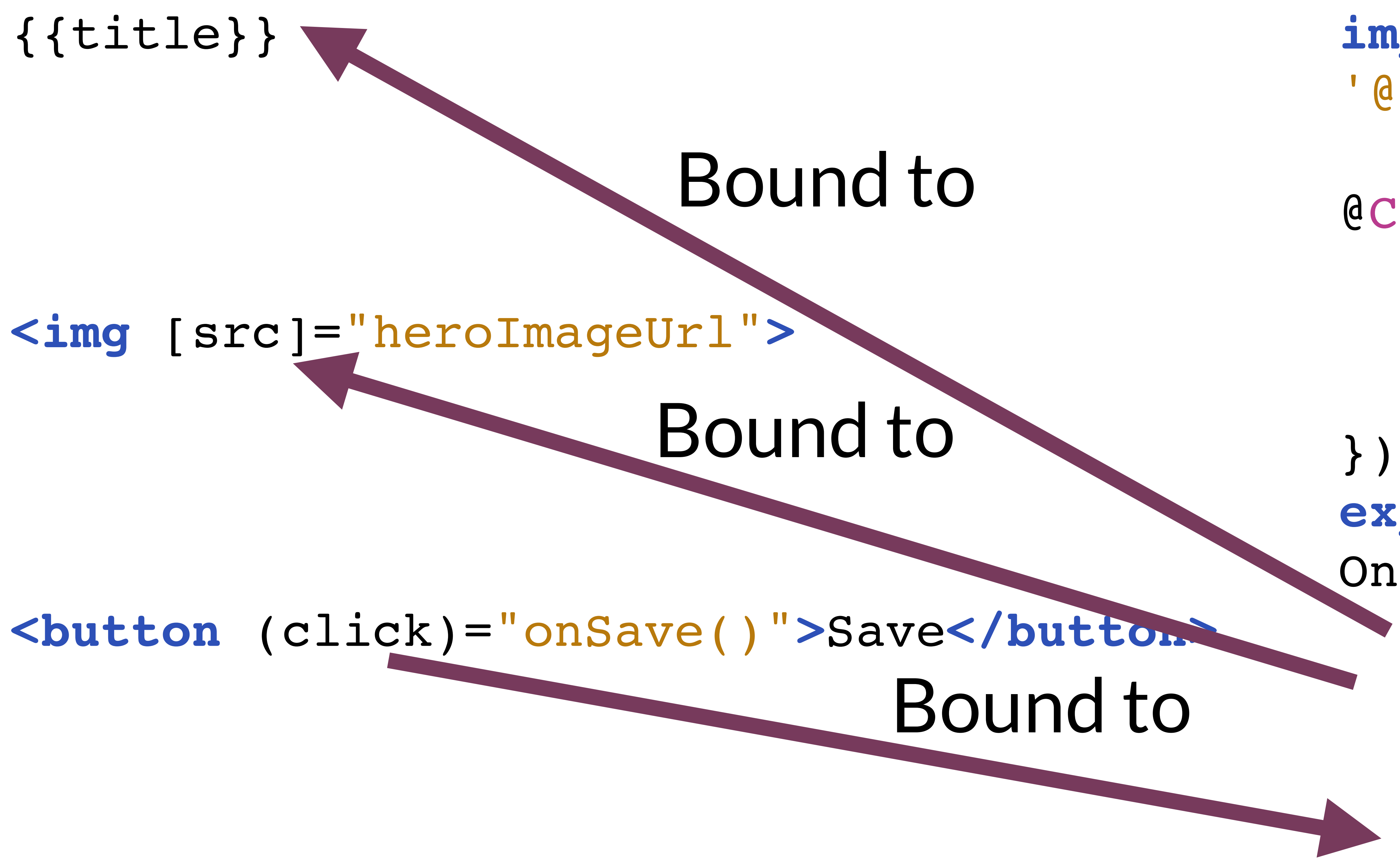
- Interpolation, property, and event are all one-way, or *read-only* binding
- For interpolation {{ }} and property [ ], binding goes from data source (.ts) to view target (.html)
- For event ( ), binding goes from view target (.html) to data source (.ts)



Data direction	Syntax
One-way from data source to view target	`{{expression}}` `[target]="expression"` `bind-target="expression"`
One-way from view target to data source	`(target)="statement"` `on-target="statement"`
Two-way	`[(target)]="expression"` `bindon-target="expression"`

https://angular.io/guide/template-syntax

One-way binding



```
{{title}}


<img [src]="heroImageUrl">


<button (click)="onSave()">Save</button>
```

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

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements
OnInit {
  title = 'example';
  heroImageUrl = 'hero.jpg';

  onSave() {
    console.log('File saved!');
  }

}
```

https://angular.io/guide/template-syntax

Two-way binding [( )]

- "You often want to both display a data property and update that property when the user makes changes"
- Most common use: binding to user-generated input
- ngModel directive enables two-way binding to input fields

```
<!--enteredText variable contains inputted text-->
<!--textChanged() is called after every keystroke-->
<input [(ngModel)]="enteredText" (change)="textChanged()">
```

https://angular.io/guide/template-syntax

Binding

```
<!--enteredText variable contains inputted text-->
<!--textChanged() is called after every keystroke-->
<input [(ngModel)]="enteredText" (change)="textChanged()">

<!--When clicked, will run the onSave() function in component.ts
file-->
<button (click)="onSave()">Save</button>

<h3>
  <!--will display the title-->
  {{title}}
  <!--will display the image at heroImageUrl-->
  <img [src]="heroImageUrl">
</h3>
```

Control flows

- @if and @for, with parentheses to identify what should be covered
- Switched from *ngIf and *ngFor in 2024, many online examples will use the old syntax

https://angular.love/diving-into-the-new-angular-control-flow-internals

@if

- Render a tag if condition is true

```
@if(isHalloween) {
  <p>
    Spooky!
  </p>
} @else {
  <p>
    Just another boring day in the neighborhood.
  </p>
}
```

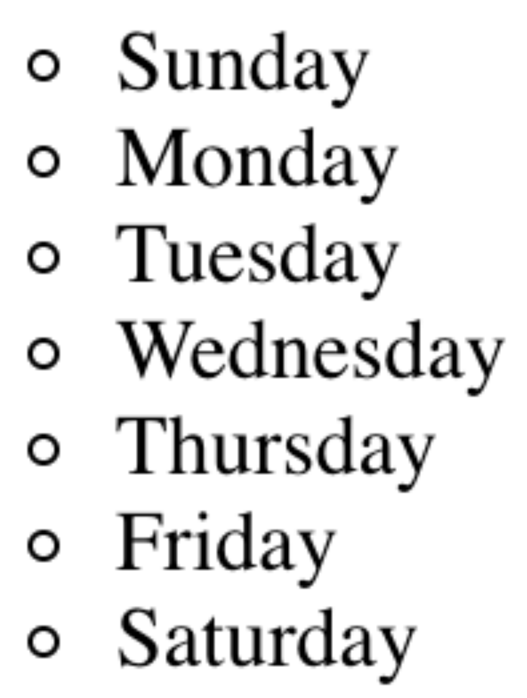
https://angular.dev/api/core/@if

@for

Subtitle

- Repeat an item multiple times

```
<ul>
  @for(day of days; track day) {
    <li>
      {{day}}
    </li>
  }
</ul>
```

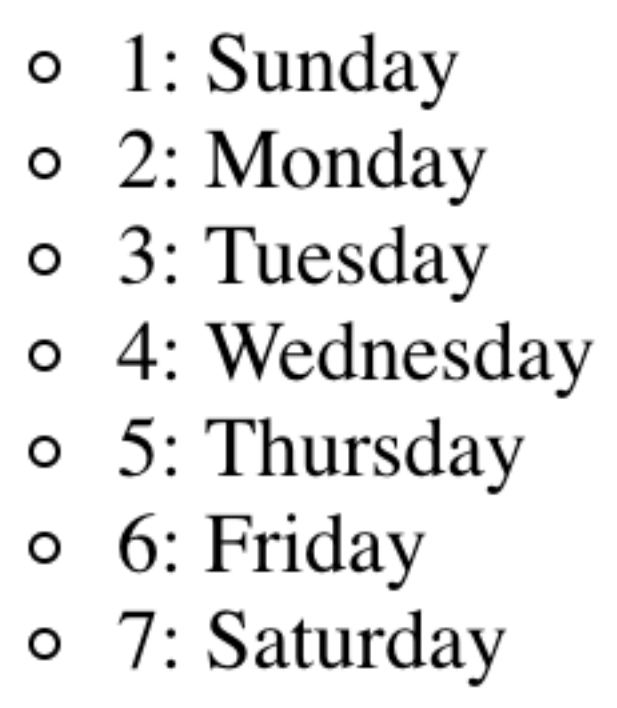


- Sunday
- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday

- "Track" is for performance, should be a unique property

- Can optionally specify index

```
<ul>
  @for(day of days; track day; let i = $index) {
    <li>
      {{i+1}}: {{day}}
    </li>
  }
</ul>
```



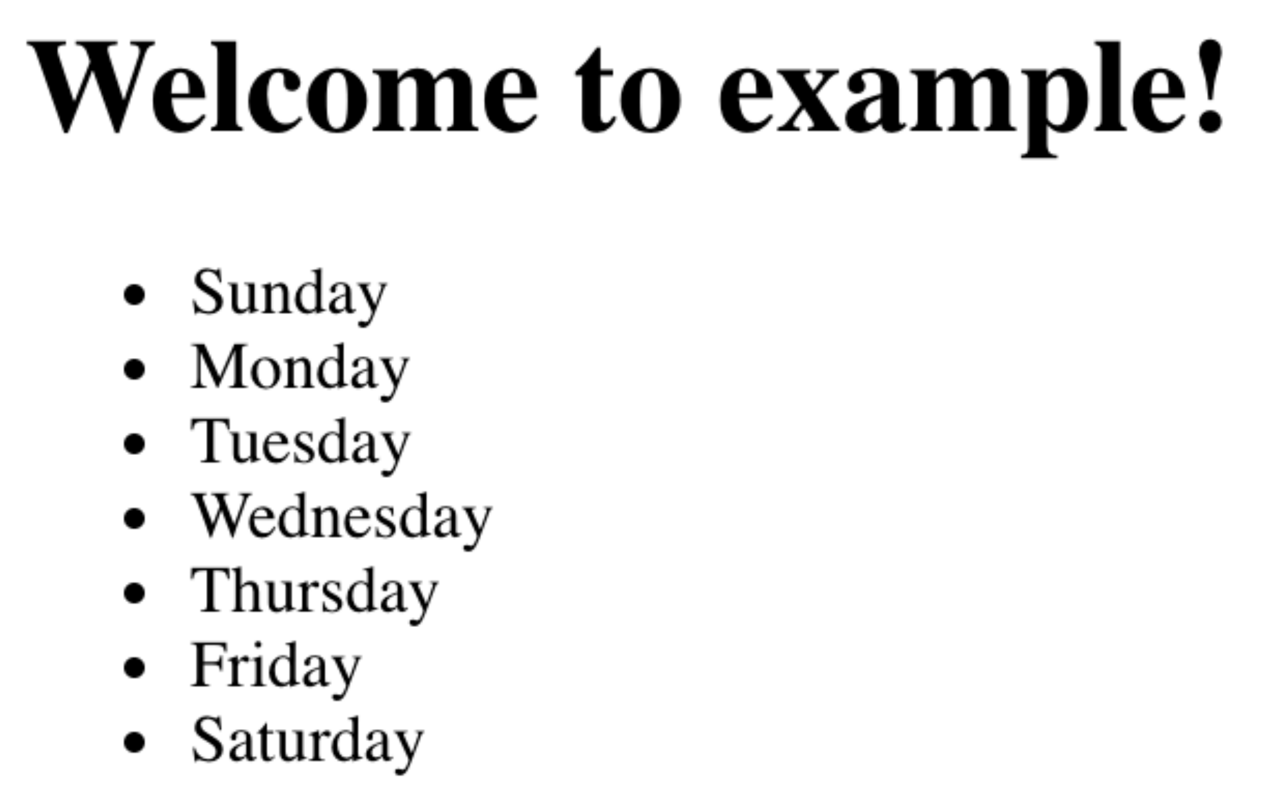
- 1: Sunday
- 2: Monday
- 3: Tuesday
- 4: Wednesday
- 5: Thursday
- 6: Friday
- 7: Saturday

https://angular.dev/api/core/@for

Using components

- Components can import other components
  - Follow the selector defined in the component's `.js` file
- In app.component.html:

```
<div>
  <h1>
    Welcome to {{ title }}!
  </h1>
  <app-day></app-day>
</div>
```

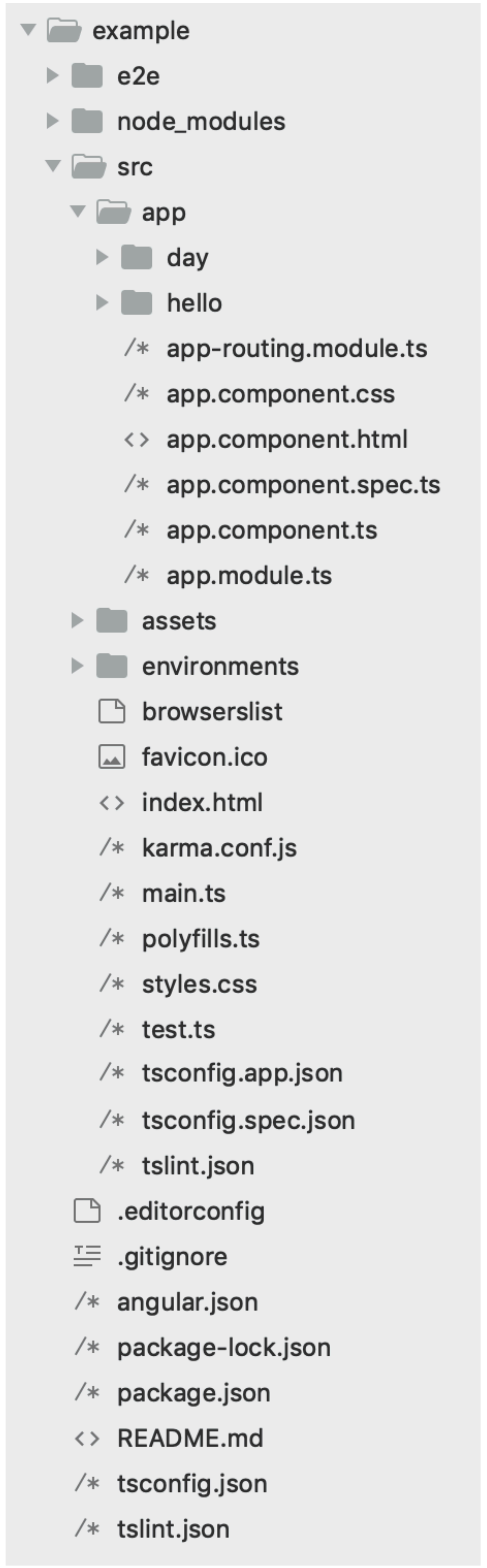


Welcome to example!

- Sunday
- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday

Angular's file structure

- Angular projects generate a *lot* of files
- Most are boilerplate, needed to set up an Angular project but rarely modified when making the interface
  - Configuration and library installation
  - Automated software testing
  - Performance and optimization



```
example
  e2e
  node_modules
  src
    app
      day
      hello
      app-routing.module.ts
      app.component.css
      app.component.html
      app.component.spec.ts
      app.component.ts
      app.module.ts
    assets
    environments
    browserslist
    favicon.ico
    index.html
    karma.conf.js
    main.ts
    polyfills.ts
    styles.css
    test.ts
    tsconfig.app.json
    tsconfig.spec.json
    tslint.json
  .editorconfig
  .gitignore
  angular.json
  package-lock.json
  package.json
  README.md
  tsconfig.json
  tslint.json
```

Other capabilities

- Interface frameworks typically offer a lot of other capabilities, including:
  - Libraries for connecting to databases
  - Plugins for native development
  - Client and server-side rendering, to optimize load times
  - Routing, or mapping URLs to your own internal structure
  - … And much more

Reflecting on interface libraries

- Building larger interfaces requires *separation* and *reusability*
  - Separation between interface elements to keep them lightweight and organized
  - Separation between files to separate data from interaction and style
  - Reusable to reduce what you need to create when you add a new element to your interface
- Interface frameworks introduce these capabilities, but have a steep learning curve

Today’s goals

By the end of today, you should be able to…

- Describe the objective of frameworks toward developing complex interfaces
- Explain a Model-View-Controller Architecture and how Angular implements the architecture
- Describe the role of an Angular component
- Broadly navigate Angular’s file structure

IN4MATX 285:
Interactive Technology Studio

Programming: Interface
Frameworks

Other concepts in Angular

Using components

- Components can specify inputs

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

@Component({
  selector: 'app-day',
  templateUrl: './day.component.html',
  styleUrls: ['./day.component.css']
})
export class DayComponent {
  @Input() today:string;
```

Input

```
  days = ["Sunday", "Monday",
"Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday"];
}
```

```
<ul>
@for(let day of days){
  <li>
    {{day}}
      @if(day == today) {
        <strong>Today!</strong>
      }
  </li>
}
</ul>
```

Input referenced in .html

Using components

- Inputs are then passed:
  - As properties if they're dynamic
  - Like any other attribute if they're static

```
<div>
  <h1>
    Welcome to {{ title }}!
  </h1>
  <app-day [today]="dayOfWeek"></app-day>
</div>
```

Sets `day` property
to `dayOfWeek` variable

```
<div>
  <h1>
    Welcome to {{ title }}!
  </h1>
  <app-day today="Friday"></app-day>
</div>
```

Sets `day` property
static value `Friday`

Using components

- Can also specify output properties

```
@Output('myClick') clicks = new EventEmitter<string>();
```

- When adding component, can specify an event to trigger when `clicks()` is called

```
<app-button (myClick)="clickMessage">click with myClick</
app-button>
```

- The event will be triggered in the parent component

```
clickMessage() {
 console.log("clicked!");
}
```

Angular routing

app-routing.module.ts (or app.routes.ts in newer Angular)

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ArtistPageComponent } from './pages/artist-page/artist-page.component';
import { TrackPageComponent } from './pages/track-page/track-page.component';
import { AlbumPageComponent } from './pages/album-page/album-page.component';
import { HomePageComponent } from './pages/home-page/home-page.component';

const routes: Routes = [
  { path: 'artist/:id', component: ArtistPageComponent},
  { path: 'track/:id', component: TrackPageComponent},
  { path: 'album/:id', component: AlbumPageComponent},
  { path: '', component: HomePageComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

⬅Listens for any endpoint `artist/:id`

`id` can be retrieved in `album-page.component.ts`

Retrieving route in a component

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-album-page',
  templateUrl: './album-page.component.html',
  styleUrls: ['./album-page.component.css']
})
export class AlbumPageComponent implements OnInit {

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
   var albumId = this.route.snapshot.paramMap.get('id');
  }

}
```

“Injecting a service”

Retrieve the id from the URI

Angular services

- Anything not associated with a specific view should be turned into a *service*
  - e.g., getting data from an API, parsing URIs for routing information
- Helps keep components lightweight
- Services can then be *injected* into a component (importing)
- To inject, import the service and retrieve it as a parameter in the constructor
- `ng generate service [name]`

Angular services

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-album-page',
  templateUrl: './album-page.component.html',
  styleUrls: ['./album-page.component.css']
})
export class AlbumPageComponent implements OnInit {

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
   var albumId = this.route.snapshot.paramMap.get('id');
  }
}
```

Importing a service

Injecting it

Service can be referenced later

Angular services

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

Defined as injectable

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

Services can inject other services!

```
@Injectable({
  providedIn: 'root'
```

What module(s) can use this service

```
})
export class SpotifyService {
 baseUrl:string = 'http://localhost:8888';

  constructor(private http:HttpClient) { }
```

HttpClient injected

```
  private sendRequestToExpress(endpoint:string) {
  }
}
```

Import a custom service

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { SpotifyService } from '../../services/spotify.service';
```

Import service via file structure

```
@Component({
  selector: 'app-album-page',
  templateUrl: './album-page.component.html',
  styleUrls: ['./album-page.component.css']
})
export class AlbumPageComponent implements OnInit {

  constructor(private route: ActivatedRoute,
private spotifyService:SpotifyService) { }
```

Inject it like any other service