Day 2 continues – npm, frameworks, dive into Angular

NPM

- Node Package Manager
- Works by looking at your package.json
- Used via CLI command `npm`
- By default - stores all the dependencies in node_modules folder inside your project
- Has local cache which is used to avoid downloads
- Dependency version resolution relies heavily on semantic versioning - see next slide for details.
- You can manage NPM versions with an NVM tool!

Semantic versioning a.k.a semver

- TLDR: all versions are formed from 3 numbers separated by dots
  - First - major version, which implies that API changes and the changes most likely WILL break things.
  - Second - minor version, which implies that API does not change or remains backward compatible. Should not break things, but in reality - a lot of things happen :)
  - Third - patch version, implies backwards compatible bug fixes.
- Feel free to read official page https://semver.org/

Yarn

- NPM's cousin
- Main difference:
  NPM installs packages sequentially
  Yarn installls packages in-parallel

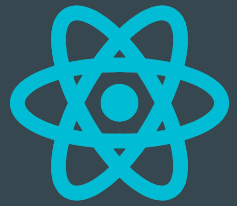
`npm i` vs `yarn`

Thorough comparison: https://phoenixnap.com/kb/yarn-vs-npm

Short framework overview

- Created by You. (Evan You).
- Very popular in the far east - China/Korea/Japan.
- View Library with supporting services.
- Promotes Modern Javascript (ES6+), though fully compatible with TypeScript.
- Supports a flavour of JSX, which allows writing CSS/JS/HTML in one file.

- Many of the plugin libraries (of which there are many) are China focused and the documentation is only available in Mandarin.
- Lack of established best practices.
- Similar forked-native and 3rd party packages are confusing.
- Many breaking changes.



- Created by Facebook.
- Most popular front end framework in the world.
- Very flexible - only deals with views
- Package size and performance can be extremely good.
- Easy to separate development between separate teams.

- Due to lack of exact standards - different developers can take entirely different approaches to building React applications.
- Have to mix and match everything.
- Configurations might be wildly different between projects, due to lack of standartization.
- No good interoperability with other frameworks
- Promotes JSX - writing CSS/HTML and JS as JS.
- Accessibility requires a lot of custom hacking.

- Created by Google.
- Model-View-Whatever framework.
- Full framework, with service coverage for most use cases.
- Promotes Typescript
- Fully interoperable with other frameworks, due to strict scoping.
- Opinionated - no need to mix and match.

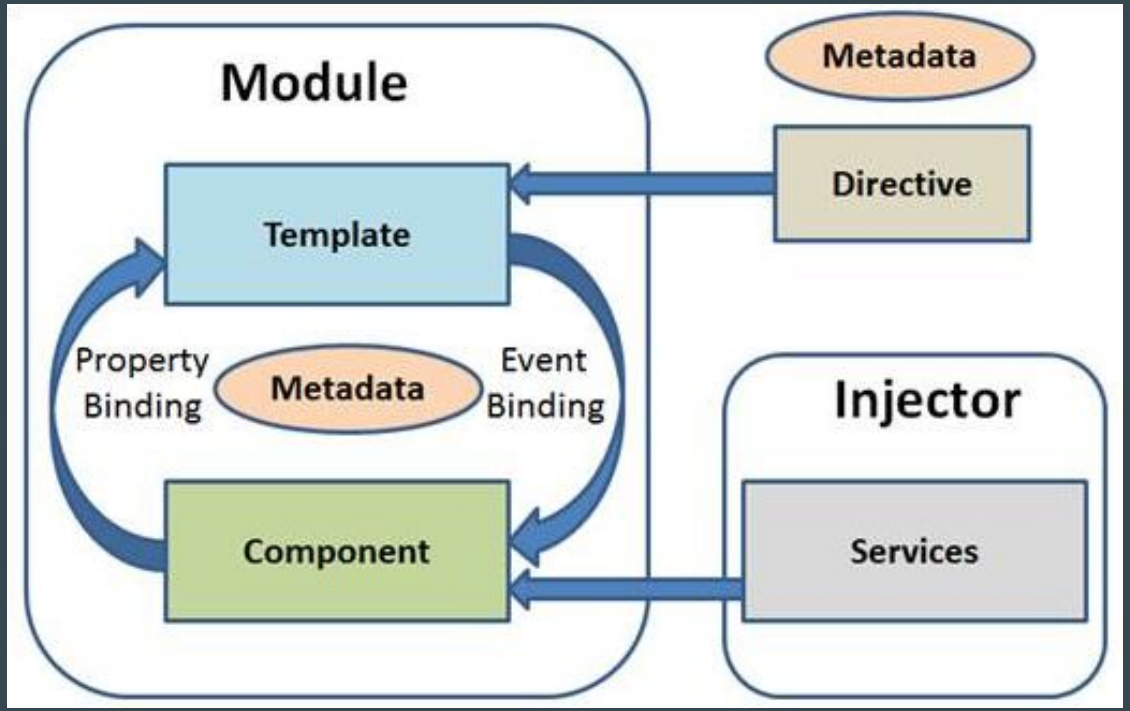
- Opinionated - hard to mix and match.
- Use of non-standard practices requires learning some new architectural patterns.
- Quite a steep learning curve for new developers.
- Hard to find information due to bad naming (Not AngularJS)

Introduction to Angular

What is Angular

- Open source typescript based framework for building client side web applications
- Created by Angular team at Google
- Angular 2+ completely different from Angular 1 (AngularJS)
- Built entirely in typescript
- ~Two major releases per year

Angular architecture

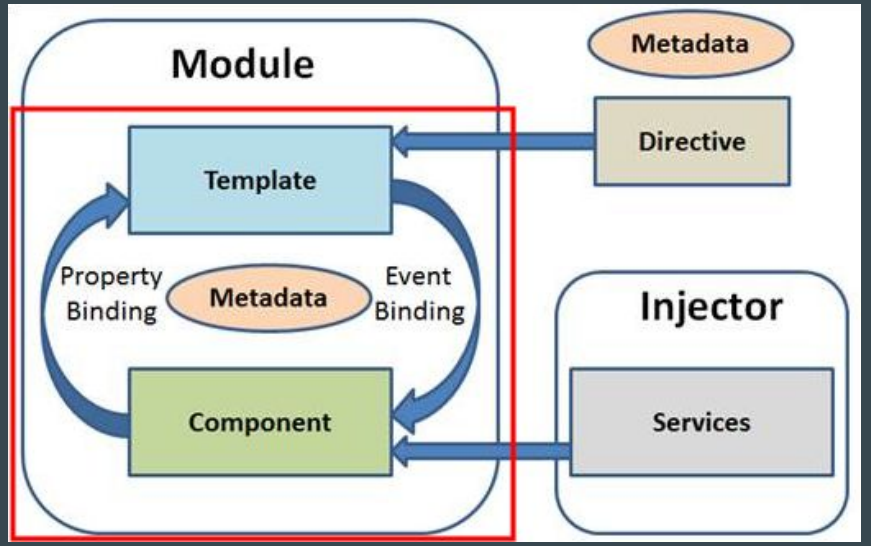


Angular architecture

- ***NgModules*** - basic Angular app building blocks, which provides compilation
- ***Component*** - a class with the template, responsible for exposing data and
- handling user-interaction logic through data binding
- **Directive** - a class that allows to transform DOM according to its instructions
- ***Service*** - a class that provides functionality not directly related to views

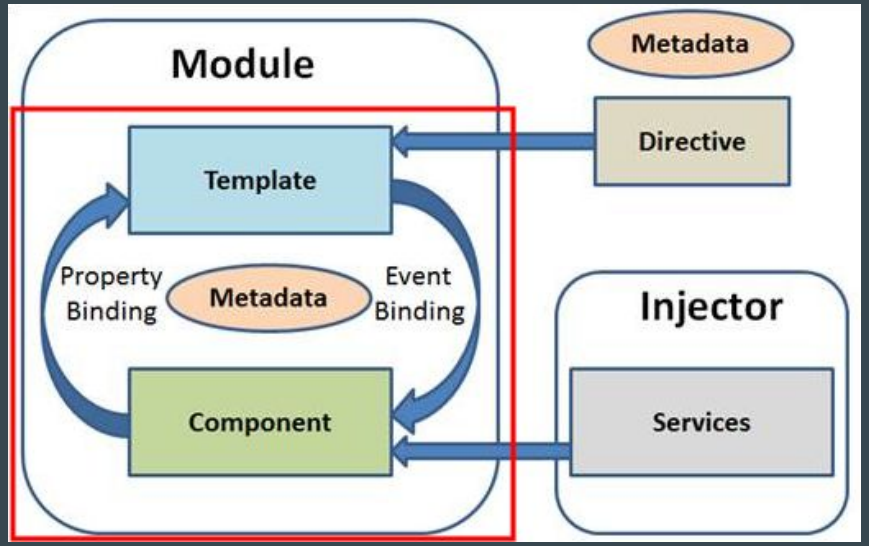
Angular module

- *@NgModule()* decorator
- Helps to structure app based on domains, workflows, etc.
- One root module (AppModule) and optional feature modules
- Can import and export functionality



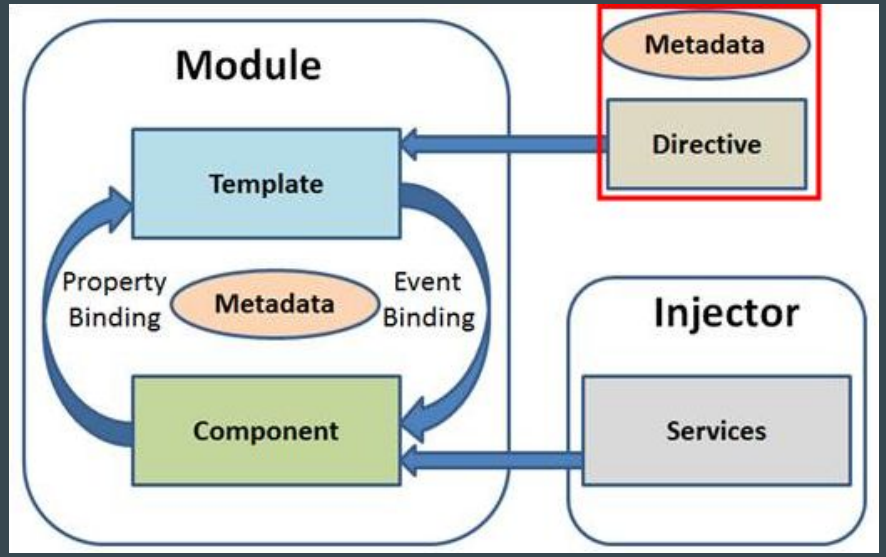
Angular component

- *@Component ()* decorator
- *Metadata* - the way to define the way of processing the class
- *Template* - component view that defines how to display the component
- *Data binding* - a process that allows apps to display data values to a user and
- respond to
- **Dependency injection** allows a class receive dependencies from another class.
- user actions



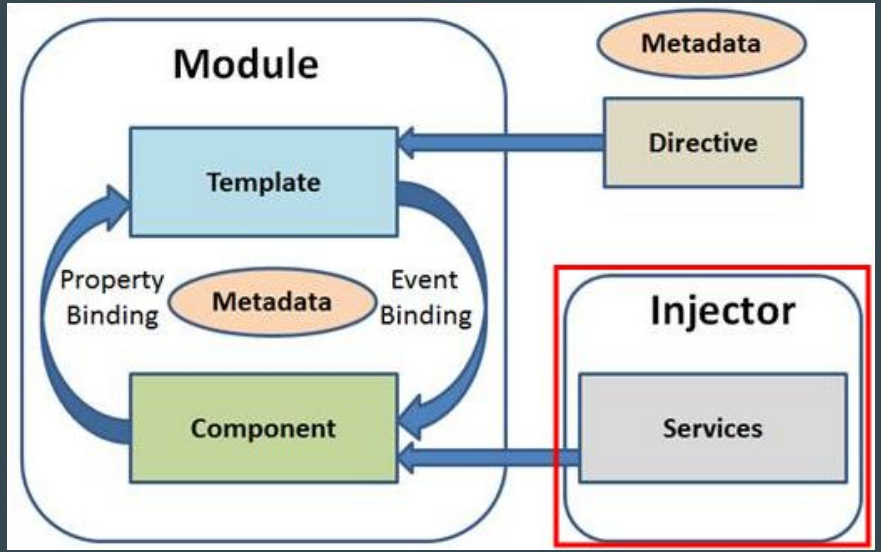
Angular directive

- *@Directive()* decorator
- Enhance element or component with additional functionality



Angular service

- *@Injectable()* decorator
- Data layer and logic
- Not component related (API requests)
- Accessed by components via dependency injection



Angular CLI

- Command-line interface tool that allows to develop and maintain Angular applications
- Reduces manual boiler plate code and saves time
- Prerequisites (for v13): Node 16.x, npm 8.x

Tasks

1. npm install -g @angular/cli – install angular on local machine
2. ng new angular-app – create angular project named ‘angular-app’
3. cd angular-app - enter project directory
4. ng serve – launches project on local machine
5. Visit http://localhost:4200/ - check that app is running properly

Dive into components

- *@Component()* decorator
- Component is the most basic UI building block of an Angular app
- The component class is marked with decorator
- Component has a lifecycle managed b Angular lifecycle hooks

```
import { Component, OnInit } from '@angular/core';
import { PaymentsService } from '../services/payments.service';
import { Payment } from '../types';

@Component({
  selector: 'app-container',
  templateUrl: './container.component.html',
  styleUrls: ['./container.component.scss']
})
export class ContainerComponent implements OnInit {
  payments: Payment[];
  payment: Payment;

  constructor(
    private paymentsService: PaymentsService,
  ) {
    this.payments = this.paymentsService.getPayments();
    this.payment = this.payments[0];
  }

  ngOnInit(): void {

  }
```

Component data binding

One-way from data source to view target: interpolation, property, attribute, class, style
One-way from view target to data source: event
Two-way view-to-source-view (often used in template forms)

```
<div (click)="handleClick($event)">
  <b>Name: </b><span>{{ payment.name }}</span>
  <br>
  <b>Amount </b><span>{{ payment.amount }}</span>
</div>
```

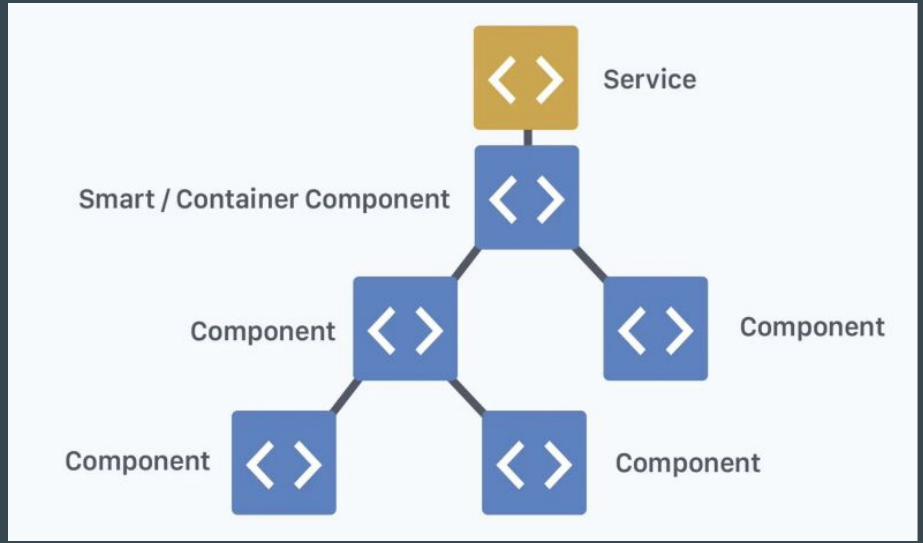
```
<h2>Container component</h2>
<app-payment [payment]="payment"
             (onPaymentClick)="handlePaymentClick($event)"></app-payment>
```

Container components and presentational components

Smart (container) component
- Communicates with services
- Render child (dumb) components

Dumb (presentational) component
- Accepts data via @Input
- Emit data changes via event

@Output
- Don’t know about services or state



Why smart/dumb components structure?

- Easy to predict and understand dumb component's behaviour
- Easier to maintain and refactor code
- Components are smaller and easier to reuse
- Easier to unit test
- Helps to avoid bugs
- Consistent developer experience, better readability

Component: smart

```ts
import { Component, OnInit } from '@angular/core';
import { PaymentsService } from '../services/payments.service';
import { Payment } from '../types';

@Component({
  selector: 'app-container',
  templateUrl: './container.component.html',
  styleUrls: ['./container.component.scss']
})
export class ContainerComponent implements OnInit {
  payments: Payment[];
  payment: Payment;

  constructor(
    private paymentsService: PaymentsService,
  ) {
    this.payments = this.paymentsService.getPayments();
    this.payment = this.payments[0];
  }

  ngOnInit(): void {

  }

  handlePaymentClick(data: Payment) {
    console.log('Payment clicked!');
    console.table(data);
  }

}
```

```html
<h2>Container component</h2>
<app-payment [payment]="payment"
             (onPaymentClick)="handlePaymentClick($event)"></app-payment>
```

Component: dumb

```ts
import ...

@Component({
  selector: 'app-payment',
  templateUrl: './payment.component.html',
  styleUrls: ['./payment.component.scss']
})
export class PaymentComponent implements OnInit {
  @Input() payment: Payment;
  @Output() onPaymentClick: EventEmitter<Payment> = new EventEmitter<Payment>();

  constructor() { }

  ngOnInit(): void {

  }

  handleClick(event: MouseEvent) {
    this.onPaymentClick.emit(this.payment);
  }

}
```

```html
<div (click)="handleClick($event)">
  <b>Name: </b><span>{{ payment.name }}</span>
  <br>
  <b>Amount </b><span>{{ payment.amount }}</span>
</div>
```

Component lifecycle hooks

- *ngOnInit() – script initialization*
- *ngOnChanges() – input data changes*
- *ngAfterViewInit() – template render*
- *ngOnDestroy() – component destruction*

Task

Generate two components with angular cli:

- Container component called ‘container’
- Presentational component called ‘payment’

```
ng generate component <componentName>
ng g c <componentName>
```

Pass data (payment details) from container component

Show passed data in presentational component

Emit events from presentational component (nameClick, amountClick)

On (amountClick) event increment payment’s amount value.

Tip: use `--dry-run` when generating to see what files will be generated/ affected

Structural directives

- Common structural directives are provided by Angular to deliver generic HTML related functionality like showing and hiding of information and iterating over data arrays to produce HTML element lists.
- *ngIf
- *ngFor

*ngIf

Used for adding or removing elements from the template based on predicate.

```
<section id="articles">
  <article *ngIf="currentArticleId === articles[0].id"></article>
</section>
```

*ngFor

Useful for iterating over arrays.

```
public articles: Article[] = [
  {
    heading: "Article 1",
    text: `Me Blimey spirits Buccane
  },
  {
    heading: "Article 2",
    text: `Hogshead bowsprit scupper
  },
  {
    heading: "Article 3",
    text: `  Starboard grog tender c
  }
];
```

```
<section id="articles">
  <article *ngFor="let article of articles"></article>
</section>
```

* and <ng-template>

- * is syntactic sugar, that is used to indicate that the element should be wrapped with <ng-template> and transformed.

```
<img *ngIf="post.img" [src]="post.img" />

Becomes

<ng-template [ngIf]="post.img">
  <img [src]="post.img" />
</ng-template>
```

```
<app-post *ngFor="let post of posts" [post]="post"></app-post>

Becomes

<ng-template ngFor let-post [ngForOf]="posts">
  <app-post [post]="post"></app-post>
</ng-template>
```

Task

1. Use *ngFor to iterate through payments in <container> element
2. Use *ngIf to render <payment> component with higher or lower specific amount value

Bonus: Directives

- *@Directive()* decorator
- Enhances element or component with functionality

```
@Directive({
  selector: '[appBackgroundColor]'
})
export class BackgroundDirective implements OnInit {
  @Input() appBackgroundColor?: string = '#333';

  constructor(private el: ElementRef) {

  }

  ngOnInit() {
    this.el.nativeElement.style.backgroundColor = this.appBackgroundColor;
    this.el.nativeElement.classList.add('appBackgroundColor');
  }

  @HostListener('click') handleClick() {
    alert('appBackgroundColor clicked!');
  }

}
```

```
<h2 [appBackgroundColor]="'#ccc'">Container component</h2>
```

Bonus: Pipes

```
<h1>{{ title | uppercase }}</h1>
```

- *@Pipe()* decorator
- Transforms provided data before it is parsed in template
- Built-in pipes: https://codecraft.tv/courses/angular/pipes/built-in-pipes/

```
<app-payment *ngFor="let payment of payments | paymentSort: key: 'name'"
             [payment]="payment"
             (onPaymentClick)="handlePaymentClick($event)"
></app-payment>
```

```
import { Pipe, PipeTransform } from '@angular/core';
import { Payment } from '../types';

@Pipe({
  name: 'paymentSort'
})
export class PaymentSortPipe implements PipeTransform {

  transform(value: Payment[], key: 'type'|'amount'|'name' = 'amount'): Payment[] {
    return value.sort( compareFn: (a :Payment , b :Payment ) => a[key] < b[key] ? -1 : 1);
  }

}
```

Examples

- git clone git@github.com:kyskiz/it-academy-angular.git
- Branches:
  - main
  - live