Angular 10 Core

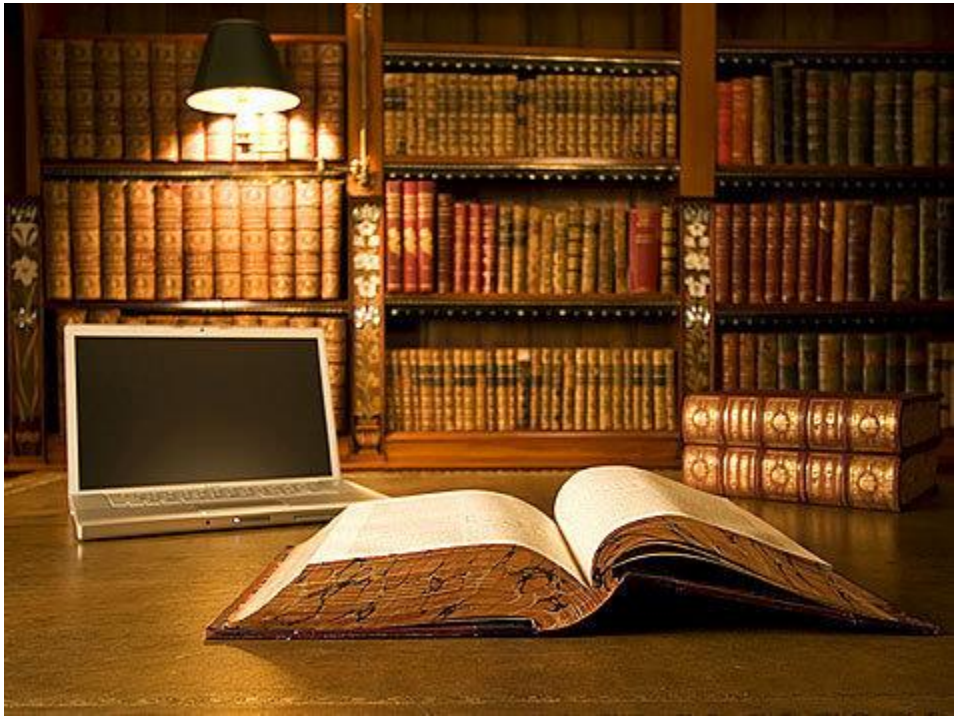
June 20th , 2020

- Sergiy Morenets, 2020

Task #10. WebStorm and Angular

1. Open created Angular project in **WebStorm** (File -> Open)
2. Review project structure and contents. Which Angular modules do you have now?
3. Set "**strictNullCheck**" property to "true" in "compilerOptions" element in tsconfig.json.
4. Open **app.module.ts** and go to the declaration of standard Angular types

Business domain



- Sergiy Morenets, 2020

Index.html


```
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Angular</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
    <app-root></app-root>
</body>
</html>
```

Bootstrap component selector (AppComponent)

app.component.html

```
<app-panel></app-panel>
<app-dashboard></app-dashboard>
```

Angular components

- ✓ Contains model, view (template) and controller (model and view update)
- ✓ Recognized by **@Component** decorator

Definition(class)

- app.component.cs

Template

- app.component.html

Styles

- app.component.css

Specification (tests)

- app.component.spec.ts

Component declaration

Global prefix

```
@Component({
  selector: 'app-root',
  template: '<div>Body</div>',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Inlined template

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

template

styles (optional)

Components. Data binding

app.component.ts

```
export class AppComponent {
  title = 'Hello, world!';

  user: object = {name: 'John'};
```

```
export class AppComponent {
  title: string;

  constructor() {
    this.title = 'Hello, world!';
  }
```

app.component.html

```
<div>{{title}}</div>
<div>{{user.name}}</div>
```

Data binding. Safe programming

app.component.ts

```
export class OrderComponent implements OnInit {
  order: Order;
```

app.component.html

```
<div>ID: {{order.id}}</div>
<div>Amount: {{order.amount}}</div>
```

```
ERROR TypeError: Cannot read property 'id' of undefined core.js:6185
    at OrderComponent_Template (order.component.html:2)
    at executeTemplate (core.js:11949)
```

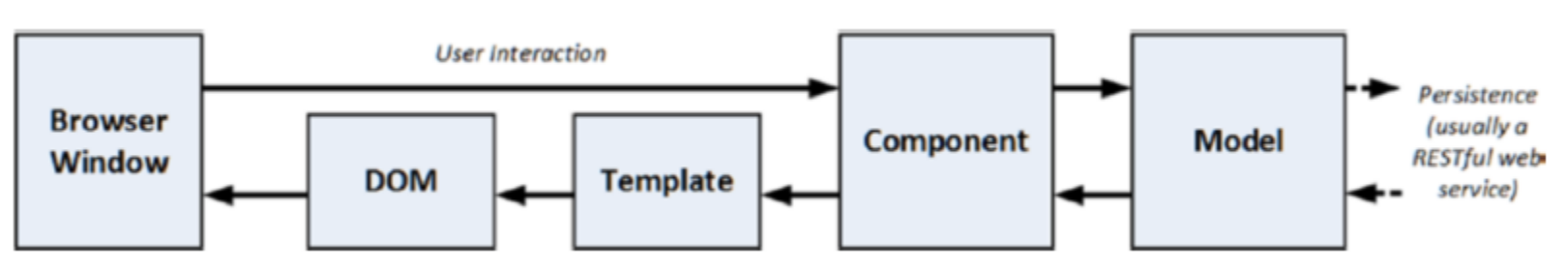
```
<div>ID: {{order?.id}}</div>
<div>Amount: {{order?.amount}}</div>
```

Angular feature

- Sergiy Morenets, 2020

Data binding

- Each HTML element is converted into corresponding DOM node
- Each element attribute has matching DOM property
- Special DOM properties can be accessed using [] syntax, for example, [textContent], [selected] or [hidden]



Two-way data binding. ngModel

```
export class OrderComponent implements OnInit {
  extendedMode: boolean;
```

One-way binding

```
<div>Extended mode: {{extendedMode}}</div>

<input [ngModel]="extendedMode">
```

```
error NG8002: Can't bind to 'ngModel' since it isn't a known property of 'input'.
```

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

```
<input [ngModel]="extendedMode">
```

One-way binding

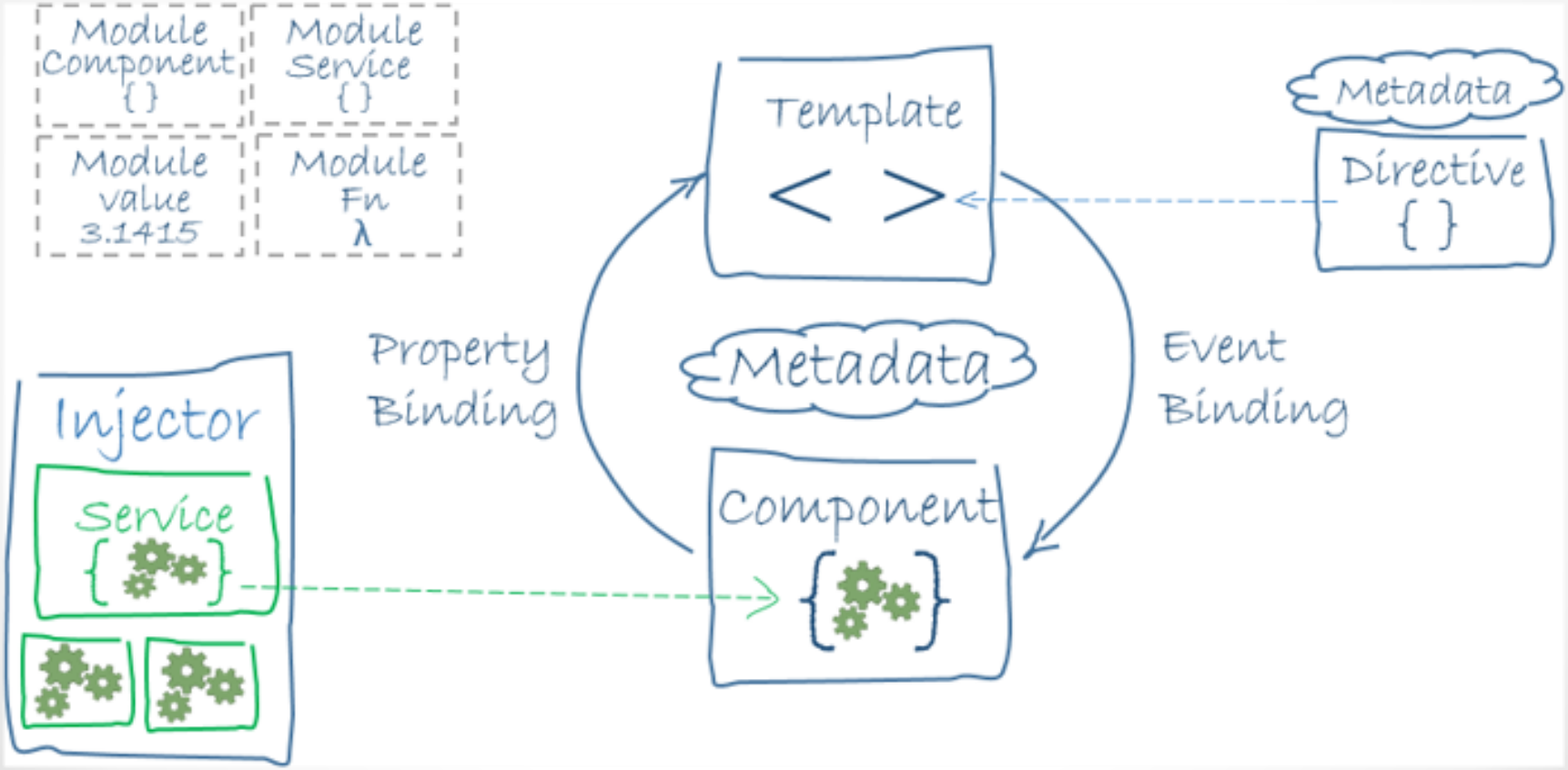
Two-way data binding. ngModel


```
<div>Extended mode: {{extendedMode}}</div>
<label> Extended mode:
  <input [(ngModel)]="extendedMode">
</label>
```

Two-way binding:

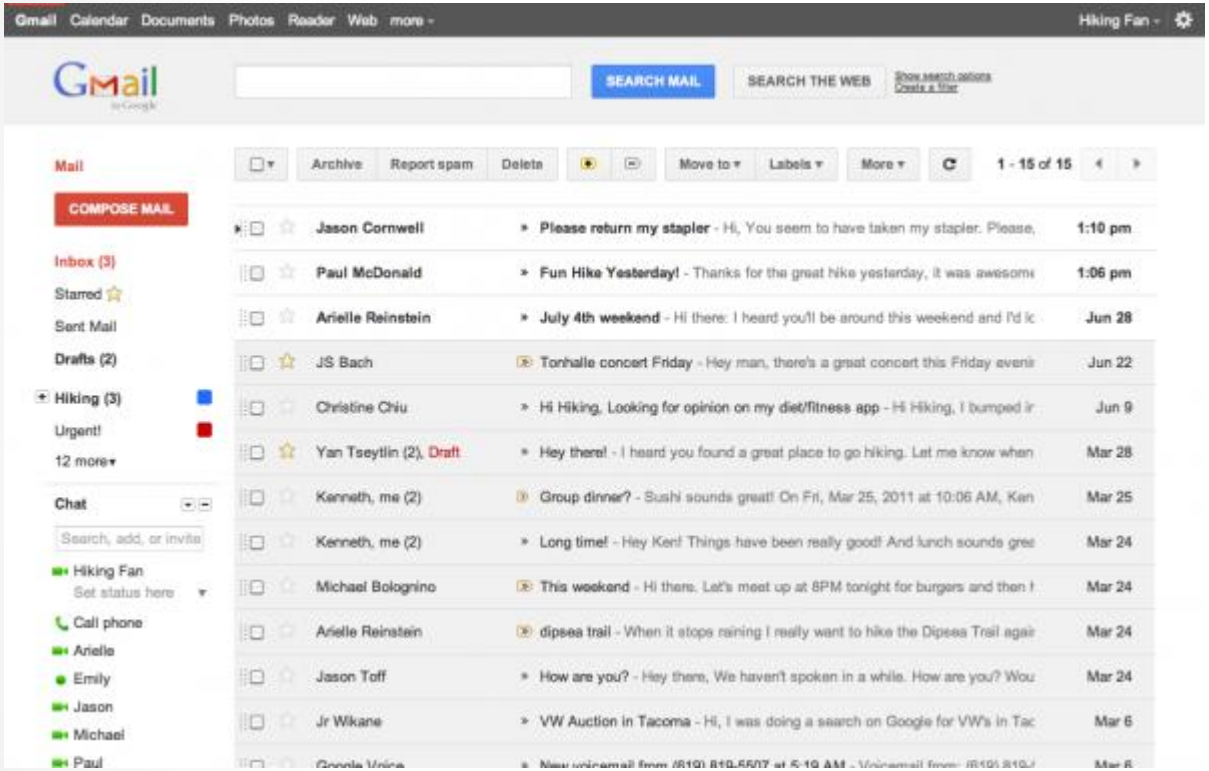
1. Model -> Template
2. Template -> Model

Angular. Building blocks



- Sergiy Morenets, 2020

Components composition



Page should be split into multiple components (tree)

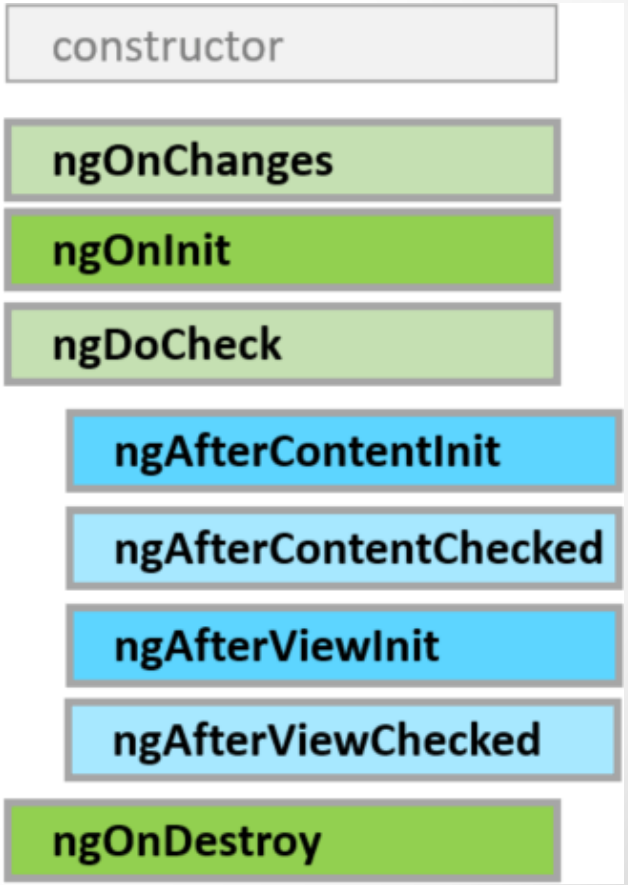
- Sergiy Morenets, 2020

Task #11. Data binding

1. Create object that stores **book** attributes (title, year, author, pages, description) in your component.
2. Update component template to display property values.
3. What will happen if you put incorrect property name in the template? Review browser console in all the cases.
4. Create new class **Book** and use it to store book properties instead of object.

Component life cycle hooks

- ✓ Lifecycle managed by Angular
- ✓ Angular creates, renders component, renders children, destroys component and removes it from DOM
- ✓ Each lifecycle hook has corresponding interface



Component life cycle hooks

Method	Description
ngOnChanges	Calls when any data-bound property changes
ngOnInit	Calls after Angular initializes input properties of component and displays template. Calls after first ngOnChanges
ngDoCheck	Calls during each change detection
ngAfterContentInit	Calls after Angular has initialized component content
ngAfterViewInit	Calls after Angular has initialized component view and its children views
ngOnDestroy	Calls before removing component from DOM. Detach any event handlers/Observable here

Event handling


```
export class AppComponent {
  log() {
    console.log('Clicked!');
  }

  logWithParams(text: string) {
    console.log(text + '.Clicked!');
  }
```

```
<button (click)="log()">Log</button>
<input type="button" value="Log more"
       (click)="logWithParams('text')"/>
```

```
<input type="text" (keydown)="log()" />
```

```
<input type="text" (keydown.control.dot)="log()" />
```

Events. Handlers

Optional parameter

```
<button (click)="handle($event)">Click me</button>
```

app.component.html

```
export class AppComponent {

    handle(event: MouseEvent): void {
        console.log(event.clientX);
        console.log(event.clientY);
        console.log(event.shiftKey);
        console.log(event.altKey);
        console.log(event.ctrlKey);
    }
```

app.component.ts

Events. Model handlers


```
export class AppComponent {
  label: string;
```

How to get notified when value has changed?

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

```
<input [(ngModel)]="label" (ngModelChange)="onChange($event)">
```

```
onChange(newValue: string): void {
  console.log('New value is ' + newValue);
}
```

Event handling. Change detection


```
@Component({
  selector: 'app-sample',
  template: '<div (click)="onClick()">{{count}}</div>'
})
export class SampleComponent {

  count = 0;

  onClick() {
    this.count++;
  }
}
```

How will Angular know that **count** has changed?

Change detection

- ✓ Angular patches all browser events, setTimeout, setInterval and AJAX requests using Zone.js library

```
function addEventListener(eventName, callback) {
    callRealAddEventListener(eventName, function() {
        callback(...);
        var changed = angular2.runChangeDetection();
        if (changed) {
            angular2.reRenderUIPart();
        }
    });
}
```

Check if template expressions have changed in the components tree

Update DOM

Task #12. Events

1. Add new **div** element at the bottom of your component.
2. Write **displayNumber** method that accepts string parameter and display number of string text characters in the created div.
3. Add event handler of **click** event for each of book text elements (title, author, description). This event handler should call displayNumber method and pass corresponding book attribute value (title, author, etc.).

Structural directives. ngIf


```
export class HomeComponent {

  hidden: boolean | undefined;

  items = [1, 2, 3];
}
```

Sign that this directive can change DOM structure

```
<div *ngIf="hidden">
  <ul>
    <li *ngFor="let item of items">{{item}}</li>
  </ul>
</div>
```

Structural directives. ngIf

```
<div *ngIf="hidden">
  <ul>
    <li *ngFor="let item of items">{{item}}</li>
  </ul>
</div>
```

```
<template [ngIf]="hidden">
  <div>
    <ul>
      <li *ngFor="let item of items">{{item}}</li>
    </ul>
  </div>
</template>
```

Angular. ngIf enhancements

```
<div *ngIf="items.length > 0; else empty_block">
  <ul>
    <li *ngFor="let item of items">{{item}}</li>
  </ul>
</div>
<ng-template #empty_block>No items</ng-template>
```

Template identifier

```
<ng-template>Ng-template</ng-template>
<ng-container>Ng-container</ng-container>
<template>Template</template>
```

Ng-container

- Sergiy Morenets, 2020

Angular. Templates


```
<!--container-->
"Ng-container"
<!--ng-container-->
▼<template _ngcontent-ooc-c13>
  #document-fragment
  "Template"
</template>
```

Angular. Structural directives


```
<div *ngIf="!hidden" *ngFor="let item of items">
  <span>{{item}}</span>
</div>
```

Can't have multiple template bindings on one element.
Use only one attribute prefixed with * (

Doesn’t reflect in DOM

```
<ng-container *ngIf="!hidden">
  <div *ngFor="let item of items">
    <span>{{item}}</span>
  </div>
</ng-container>
```

Structural directives. Case


```
<ul>
  <li *ngfor="let item of items">{{item}}</li>
</ul>
```

Can't bind to 'ngforOf' since it isn't a known property of 'li'

Structural directives. ngStyle


```
<div [style.color]="'red'">Text</div>
<div [ngStyle]="{color : 'red'}">New text</div>
```

```
<style>
   .styleRed {
       color: red;
   }
</style>
```

```
export class AppComponent {
   red: boolean;
}
```

```
<div [ngClass]="{styleRed: red}">Colored</div>
<input type="checkbox" [(ngModel)]="red"/>
```

Structural directives. ngSwitch


```
export class HomeComponent {

  category = 'Food';

<div [ngSwitch]="category">
  <span *ngSwitchCase="'Food'">Food</span>
  <span *ngSwitchCase="'Devices'">Devices</span>
</div>
```

Task #13.Structural directives


```
<div>
    <div class="javascript">
        <h4>JavaScript: The Good Parts</h4>
        <div>Author: Douglas Crockford</div>
        <div>Year: 2008</div>
        <div>Pages: 172</div>
        <div>Description: This authoritative book
    </div>
    <div>
        <h4>Mastering TypeScript</h4>
        <div>Author: Nathan Rozentals</div>
        <div>Year: 2015</div>
        <div>Pages: 364</div>
        <div>Description: Build enterprise-ready,
    </div>
```

Task #13. Structural directives

1. Review **sample.html** from **Project/Task13**.
2. Update component and component template so that it produces the same HTML as with sample.html.

Services

- Services provide common and reusable functionality for the entire application (components, directives, pipes)
- Allows to isolate functionality and increase flexibility
- Allows to share data between building blocks of the project

Dependency injection

- ✓ Design pattern
- ✓ Avoids tight coupling
- ✓ Separate responsibilities between components
- ✓ Increases code flexibility and reusability

DI. Declare & use service


```
@Injectable()
export class OrderService {

  getOrderTypes(): string[] {
    return ['CASH', 'CARD'];
  }
}
```

Required since Angular 9

Instance automatically created by Angular

```
export class OrderComponent {

  private orderTypes: string[];

  constructor(orderService: OrderService) {
    this.orderTypes = orderService.getOrderTypes();
  }
}
```

Automatically wired and injected by Angular

Service providers

Provider	Description
Class provider	Angular creates instance of a class and uses it for DI
Value provider	Angular uses provided object to resolve DI
Factory providers	Angular uses provided function to resolve DI
Existing service provider	Angular uses alias of another service to resolve DI

DI. Class provider

Register service using class provider

```
  providers: [OrderService],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

```
export class OrderComponent implements OnInit {
  private orderTypes: string[];

  constructor(private orderService: OrderService) {
  }

  ngOnInit(): void {
    this.orderTypes = this.orderService.getOrderTypes();
  }
```

Declare service as class property

DI. Choose class service


```
@Injectable()
export class DynamicOrderService {

  getOrderTypes(): string[] {
    // return from dynamic source
```

The same functions as in OrderService

```
  providers: [{provide: OrderService, useClass: OrderService}],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

```
providers: [{
  provide: OrderService, useClass:
    environment.production ? DynamicOrderService : OrderService
}],
```

DI. Choose class service


```
@Component({
  selector: 'app-order',
  templateUrl: './order.component.html',
  providers: [{provide: OrderService, useClass: DynamicOrderService}]
})
export class OrderComponent implements OnInit {
```

Dynamically replace
injected service on component level

Services & DI


```
@Injectable()
export class OrderService {

  getOrderTypes(): string[] {
    return ['CASH', 'CARD'];
  }
}
```

```
export class OrderComponent {
  constructor(orderService: OrderService) {
```

NullInjectorError: No provider for AppService!

```
export class OrderComponent {
  constructor(@Optional() orderService: OrderService) {
    if (orderService) {
      // TODO use orderService
```

Services & DI. Angular 6

Tree shaking providers

```
@Injectable({
  providedIn: 'root'
})
export class OrderService {

  getOrderTypes(): string[] {
    return ['CASH', 'CARD'];
  }
}
```

Name of the module to provide service

Task #14. Services and DI

1. Create new service **BookService** that will return list of books. You can use Angular CLI command: *ng g s <service_name>.*
2. Inject it into your component so your component will take languages and display it on the page.
3. What is the principal difference between service and component?
4. Try to register service using [providers] attribute of NgModule.

Pipes. Preconditions


```
export class AppComponent {

  upperCase(text: string): string {
    return text.toUpperCase();
  }
}
```

Required in multiple components

```
<div>{{ upperCase( text: 'text') }}</div>
```

Not resolvable in template

```
export class TextUtils {
  static upperCase(text: string): string {
    return text.toUpperCase();
  }
}
```

Pipes

- ✓ Used to transform and filter data
- ✓ Raw data formatting
- ✓ Called ‘filters’ in Angular 1.x
- ✓ Can be used in HTML/application code
- ✓ No orderBy/sanitize pipes

Built-in pipes

Name	Description	Example
json	Converts object into JSON text fomat	{{book | **json**}}
slice	Filters collection(array) creating new sub-collection	*ngFor="let book of books | **slice** : 0: 5"
number	Formats number using current regional settings	{{ amount | **number** }} {{ amount | **number** : '.2-2'}}
percent	Format number into percentage format	{{ amount | **percent** }}
currency	Applies currency symbol to the number	{{ amount | **currency** }} {{ amount | **currency** : 'UAH'}} {{ amount | **currency** : 'EUR' : 'symbol'}}

Built-in date/time formatting

Example	Result
{{ eventDate | date : '**longDate**'}}	May 15, 2017
{{ eventDate | date : '**shortDate**'}}	5/15/2017
{{ eventDate | date : '**fullDate**'}}	Monday, May 15, 2017
{{ eventDate | date : '**yyyy-MM-dd**'}}	2017-05-15
{{ eventDate | date : '**shortTime**'}}	11:10 AM
{{ eventDate | date : '**mm:ss**'}}	10:40

```
eventDate: Date = new Date();
```

Pipes and DI

app.module.ts

```
  bootstrap: [AppComponent],
  providers: [JsonPipe]
})
export class AppModule {
}
```

```
export class PanelHeaderComponent {
  constructor(jsonPipe: JsonPipe) {
    console.log(jsonPipe.transform({'data' : 1}));
  }
```

Pipes declarations and usage


```
@Pipe({name: 'stub' })
export class StubPipe implements PipeTransform {

  transform(value: any, ...args: any[]): any {
    return value;
  }
}
```

List of arguments (optional)

Example of functional programming

Pipe chaining

```
{{amount | stub | currency : 'EUR' : 'code' }}
```

- Sergiy Morenets, 2020

Task #15. Pipes

1. Add new text box that will store the number of books to display on the page.
2. Add pipe **slice** to the **ngFor** directive so that you can limit the number of displayed books.
3. Add new property **price** to the **Book** class. Update your book component template to display book price.
4. Create new pipe that will **quote** the text and apply it to the book title.

Forms

- Uses reactive(code-driven) or declarative(template) way
- Imports ReactiveFormsModule or FormsModule modules
- Allows better controller of data initialization, validation and submission

Declarative forms. Component


```
export class RegistrationComponent {

  user = new User();

  register(): void {
    console.log('Saving user : ' +
      JSON.stringify(this.user));
  }
}
```

```
export class User {
  login: string;

  password: string;
}
```

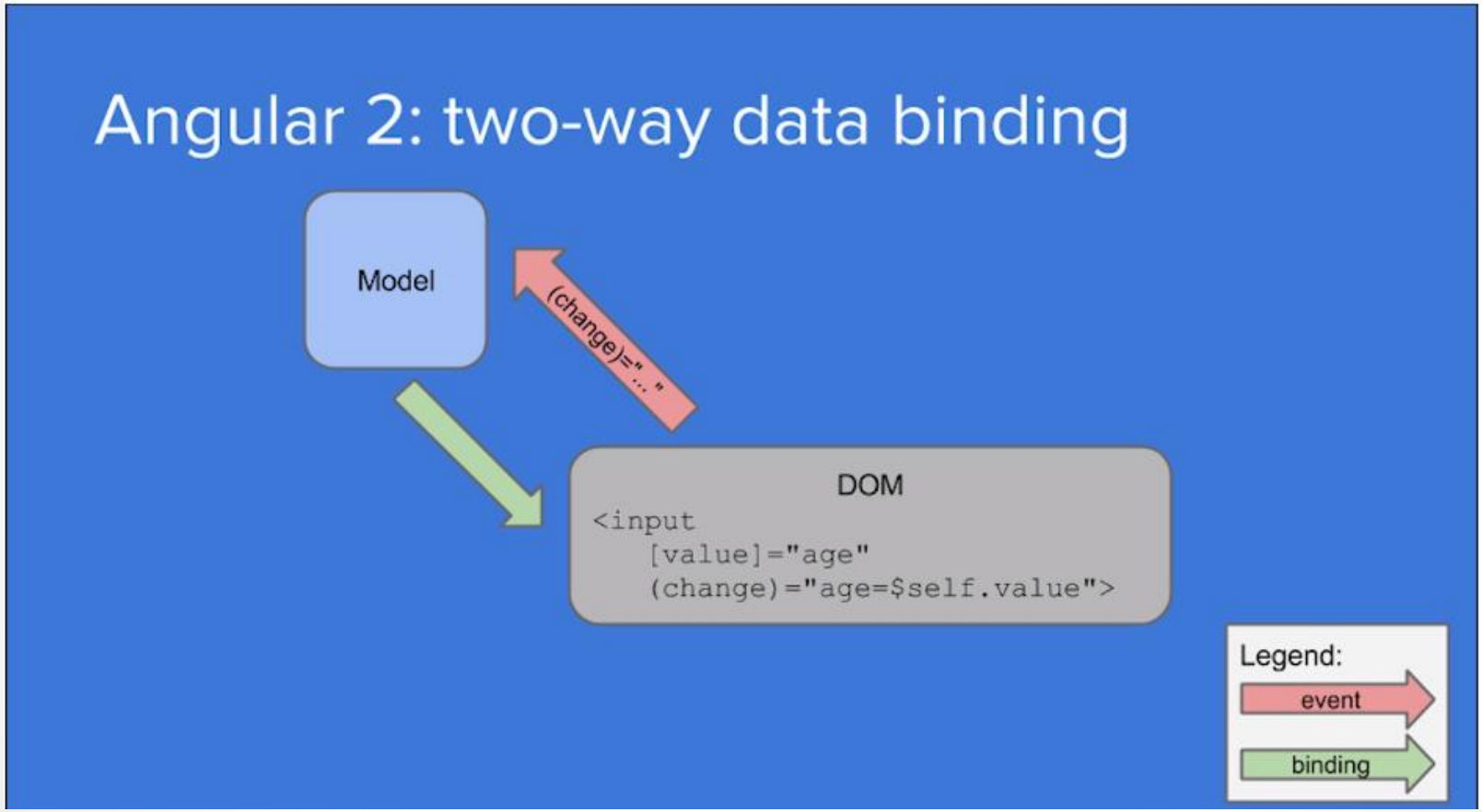
- Sergiy Morenets, 2020

Declarative forms. Templates

Imported from FormsModule

```
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input name="username"
                                  [(ngModel)]="user.username">
  </div>
  <div>
    <label>Password</label><input type="password"
                                  name="password" [(ngModel)]=
    "user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

Two-way binding



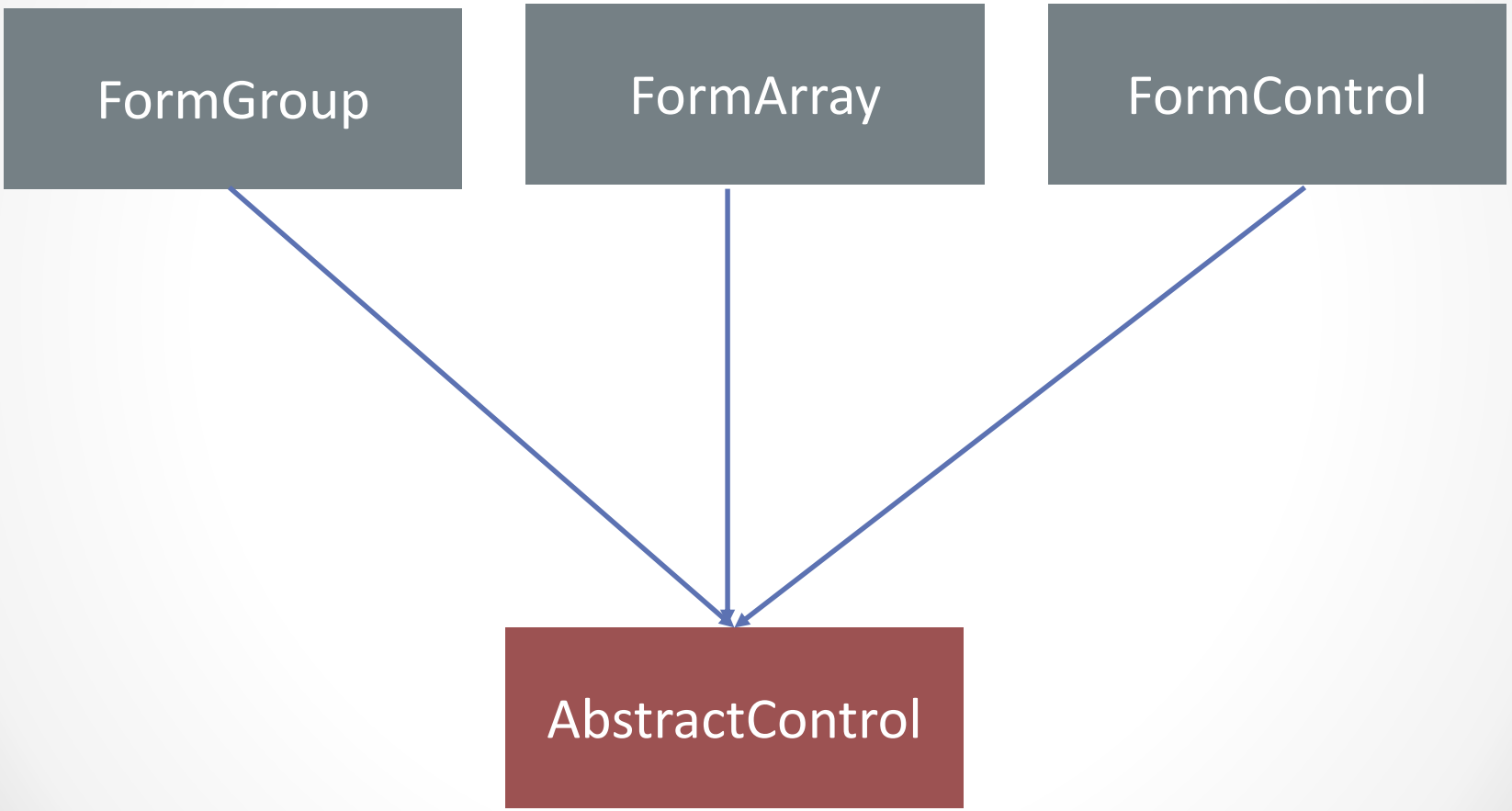
- Sergiy Morenets, 2020

Declarative forms. Validation


```
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input name="username"
        required minlength="2" [(ngModel)]="user.username">
  </div>
  <div>
    <label>Password</label><input type="password"
        required name="password" [(ngModel)]=
    "user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

- Sergiy Morenets, 2020

FormsModule types



- Sergiy Morenets, 2020

Validation. FormControl

Attribute	Description
valid	True if pass all the validations
errors	Field errors
dirty	True once someone changed the value
pristine	True until someone modified the value
touched	True if element has got focus
untouched	True if element hasn't got focus
value	Field value
hasError	Check for specific error

Templates. Local variables

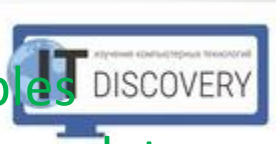
Local template variable

```
<form>Name: <input #name/>
  <button (click)="name.value = 'Sample value'">
    Change value
  </button>
</form>
```

```
accept(value: string) {
  console.log('Value is ' + value);
}
```

```
<form>Name: <input #name/>
  <button (click)="accept(name.value)">
```

Declarative forms. Validation



Temporary variables available in this template

FormControl

FormGroup

```
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username"
          required minlength="2" ngModel #username="ngModel">
  </div>
  <div *ngIf="username.dirty && !username.valid">Please, fill in
    username</div>
  <div>
    <label>Password</label><input type="password"
          ngModel
          required name="password" #password="ngModel">
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register</button>
</form>
```

Declarative forms. Component


```
export class RegistrationComponent {

  register(user: User): void {
    console.log('Saving user : ' +
      JSON.stringify(user));
  }
}
```

{ “username” : “John”,
“password”: “123”}

Task #16. Template-driven forms

1. Use angular-cli command “**ng generate component <component_name>”** to generate your component.
2. This component should contain form with input elements to enter book attributes and submit button.
3. Add basic validation to the form elements.
4. Use this component for bootstrapping in **NgModule** directive. Add it to the index.html.
5. Check that submission and validation works.

Reactive forms

- ✓ Imperative or code-driven approach
- ✓ Allows to dynamically create controls
- ✓ You need to import **ReactiveFormsModule** in your module
- ✓ Provide asynchronous API for tracking changes

Component. Reactive forms

Control name

```
@Component({
  selector: 'app-product',
  template: 'Name:<input type="text" [formControl]="name">'
})
export class ProductComponent implements OnInit {
  name: FormControl;

  constructor() {
    this.name = new FormControl('');
  }
```

@angular/forms

Initial component value

Reactive forms. Simplified builder


```
export class LoginComponent implements OnInit {
  userForm: FormGroup;

  constructor(formBuilder: FormBuilder) {
    this.userForm = formBuilder.group({
      username: '',
      password: ''
    });
  }
```

@angular/forms

Initial component value

{“username”: “”, “password”: “”}

```
register() {
  console.log('Saving user: ' +
    JSON.stringify(this.userForm.value));
}
```

FormGroup class

Method	Description
addControl(control)	Add this control to the group
removeControl(control)	Removes this control from the group
setValue(obj)	Assigns value to the form. If some attributes are missing then error is thrown
patchValue(obj)	Copies properties from the object without any errors
reset()	Reset controls values and state

Components & validation


```
export class LoginComponent implements OnInit {
  userForm: FormGroup;

  constructor(formBuilder: FormBuilder) {
    this.userForm = formBuilder.group({
      username: formBuilder.control('', [Validators.required,
        Validators.minLength(3), this.validateEmail]),
      password: formBuilder.control('', Validators.required)
    });
  }
```

Form name

Control unique name

Built-in or custom validators

Reactive forms. Validators

Validator	Description
min(limit)	Validates that control value is less then limit
max(limit)	Validates that control value is greater then limit
required	Validates that control has non-empty value
requiredTrue	Validates that control has true value
email	Validates that control value is email
minLength(length)	Validates that control value has minimum length
maxLength(length)	Validates that control value has maximum length
pattern(text)	Validates that control value matches regular expression
compose	Joins multiple validators

Reactive forms. Template


```
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label>
    <input formControlName="username">
  </div>
  <div>
    <label>Password</label>
    <input type="password" formControlName="password">
  </div>
  <button type="submit">Register</button>
</form>
```

Form name

Control name

No feedback to user about validation errors

Validation. Reactive forms


```
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label>
    <input formControlName="username">
    <div *ngIf="userForm.get('username').hasError('required')">
      Please, fill in Username
    </div>
  </div>
  <div>
    <label>Password</label>
    <input type="password" formControlName="password">
  </div>
  <button type="submit" [disabled]="!userForm.valid">Register
  </button>
</form>
```

Validator type

Check control errors

Check form state

Data binding


```
export class LoginComponent implements OnInit {
  userForm: FormGroup;

  user: User;

  constructor(formBuilder: FormBuilder) {
    this.userForm = formBuilder.group({
      username: '',
      password: ''
    });
  }
ngOnInit(): void {
  this.userForm.patchValue(this.user);
}
```

```
class User {
  username: string;
  password: string;
}
```

Copies data to form controls

Task #17. Reactive forms

1. Review **BookRegistrationComponent**.
2. Convert template-driven forms into reactive forms.
3. Update template and add form name and form control names
4. Check that submission and validation works.
5. (**Optional**) Clear input values and “touched” state after submission.

Angular. Directives

- Component without template
- Annotated with **@Directive**
- Attach behavior to the elements of DOM
- Selectors can be of element(**panel**), class**(.error**) or attribute**([text])** type

Directives


```typescript
@Directive({
  selector: '[appLogger]'
})
export class LoggerDirective {

  constructor() {
    console.log('Logger initialized');
  }
```

logger.directive.ts

```html
<div appLogger></div>
```

app.component.html

Attach directive to DOM element

Directives. Inputs


```
@Directive({
  selector: '[appLogger]',
  inputs: ['logText']
})
export class LoggerDirective {

  set logText(text: string) {
    console.log('Logger initialized with: ' + text);
  }
```

List of input parameters

```
<div appLogger
  [logText]="'random'"></div>
```

```
@Directive({
  selector: '[appLogger]'
})
export class LoggerDirective {
  @Input() logText: string;
```

Directives. Parameters initialization


```
export class LoggerDirective implements OnInit {
  @Input() logText: string;

  constructor() {
    console.log('Logger created');
  }

  ngOnInit(): void {
    console.log('Logger initialized with text: ' + this.logText);
  }
```

Initialized before ngOnInit

Directives. Attributes

```
@Directive({
  selector: '[appLogger]'
})
export class LoggerDirective implements OnInit {
  @HostBinding('style.backgroundColor')
  @Input()
  bgColor = 'red';
```

```
<div appLogger [bgColor]="'red'">Header</div>
```

Decorate functionality of existing component

Change background color of attached **div** element

Directives. Events

```
export class LoggerDirective implements OnInit {
  @HostBinding('style.backgroundColor')
  @Input()
  bgColor = 'red';

  @HostListener('mouseenter')
  mouseEnter() {
    this.bgColor = 'blue';
  }

  @HostListener('mouseleave')
  mouseLeave() {
    this.bgColor = 'yellow';
  }
```

Change bgColor when
Mouse enter/leaves attached element

Directives. Binding DOM element


```
constructor(private el: ElementRef) {
}
```

Reference to the attached element

Injected via DI

DOM element

```
ngOnInit(): void {
  this.el.nativeElement.querySelector('.class1')
                .classList.add('column_visible');
  this.el.nativeElement.querySelector('.class2')
                .style.display = 'block';
}
```

Task #18. Directives

1. Create new directive to your project.
2. Add necessary properties for the directives. You can try **inputs** attribute and **@Input** annotation for that purpose.
3. Implement **OnInit** interface and add ngOnInit method in your directive.
4. Add this directive to the **declarations** block of our module.
5. Add this directive to your component and make sure it works.

Components. Types

Presentational

- No interaction with services
- Stateless
- Input/Output usage
- Easy to test/reuse

Container

- Combine presentational components
- Maps application state
- Interact with services
- Handle events

Components. Inputs


```
@Component({
  selector: 'app-status',
  templateUrl: './status.component.html'
})
export class StatusComponent implements OnInit {

  @Input()
  status: string;

  ngOnInit() {
    console.log('Status text is ' + this.status);
  }
}
```

```
<app-status [status]="'OK'"></app-status>
```

Components. Inputs and aliases


```
export class StatusComponent implements OnInit {

  @Input('current-status')
  displayed: boolean;
```

```
<app-status [current-status]="'Error'"></app-status>
```

Components. Type checking


```
export class StatusComponent implements OnInit {

  @Input()
  displayed: boolean;
```

```
<app-status [displayed]="true"></app-status>
```

Compilation error

```
<app-status [displayed]="aaa"></app-status>
```

String parameter

```
<app-status [displayed]="'true'"></app-status>
```

Angular compiler options

- Options for Angular AOT compiler (template compiling)

Option	Description
enableIvy	Use Ivy Renderer (true) or View Engine
fullTemplateTypeCheck	Enables the binding expression validation phase
preserveWhitespaces	When false, removes blank text nodes from compiled templates
strictInjectionParameters	When true, reports an error if type of injected parameter is unknown
strictTemplates	When true, enables strict template type checking. Only available when using Ivy (Angular 9).

Angular compiler. Strict templates

- ✓ Checks the pipes return type
- ✓ Checks the type of references to directives/pipes
- ✓ Verifies component/directive type binding (including generics)
- ✓ Enables type checking in ngFor and ngIf (embedded views)
- ✓ Verifies type of $event

```
"angularCompilerOptions": {
  "strictTemplates": true,
  "strictInjectionParameters": true
}
```

tsconfig.json

Angular compiler. Strict templates


```ts
export class StatusComponent implements OnInit {

  @Input()
  displayed: boolean;
```

```html
<app-status [displayed]="'true'"></app-status>
```

```
ERROR in src/app/app.component.html:4:13 - error TS2322:
Type 'string' is not assignable to type 'boolean'.

4 <app-status [displayed]="'true'"></app-status>
              ~~~~~~~~~~~~~~~~~~~~
```

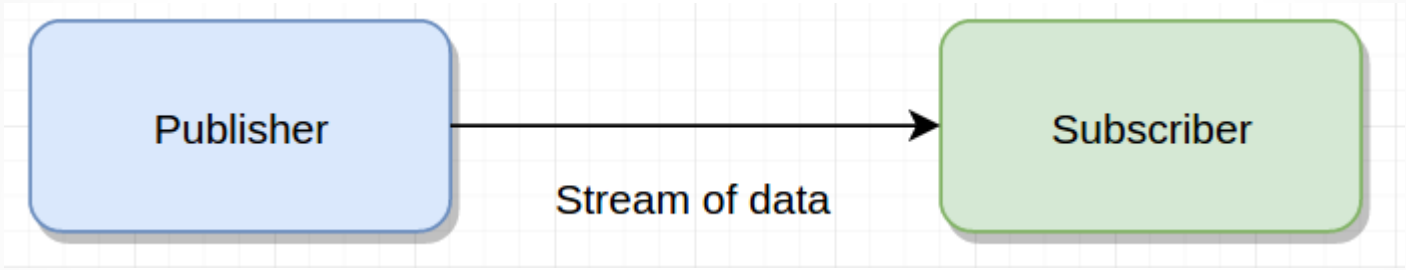
ng build

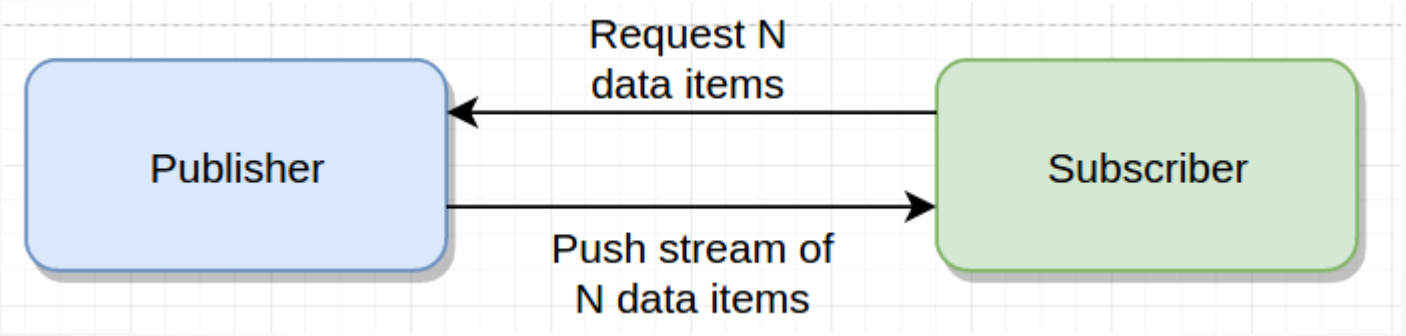
- Sergiy Morenets, 2020

Task #19. Components inputs

1. Create new component that will display single book in the list. Add necessary input element(s).
2. Embed new component into template of the existing container component. Supply necessary attribute values.
3. Run application and make sure it works properly.

Reactive streams





- Sergiy Morenets, 2020

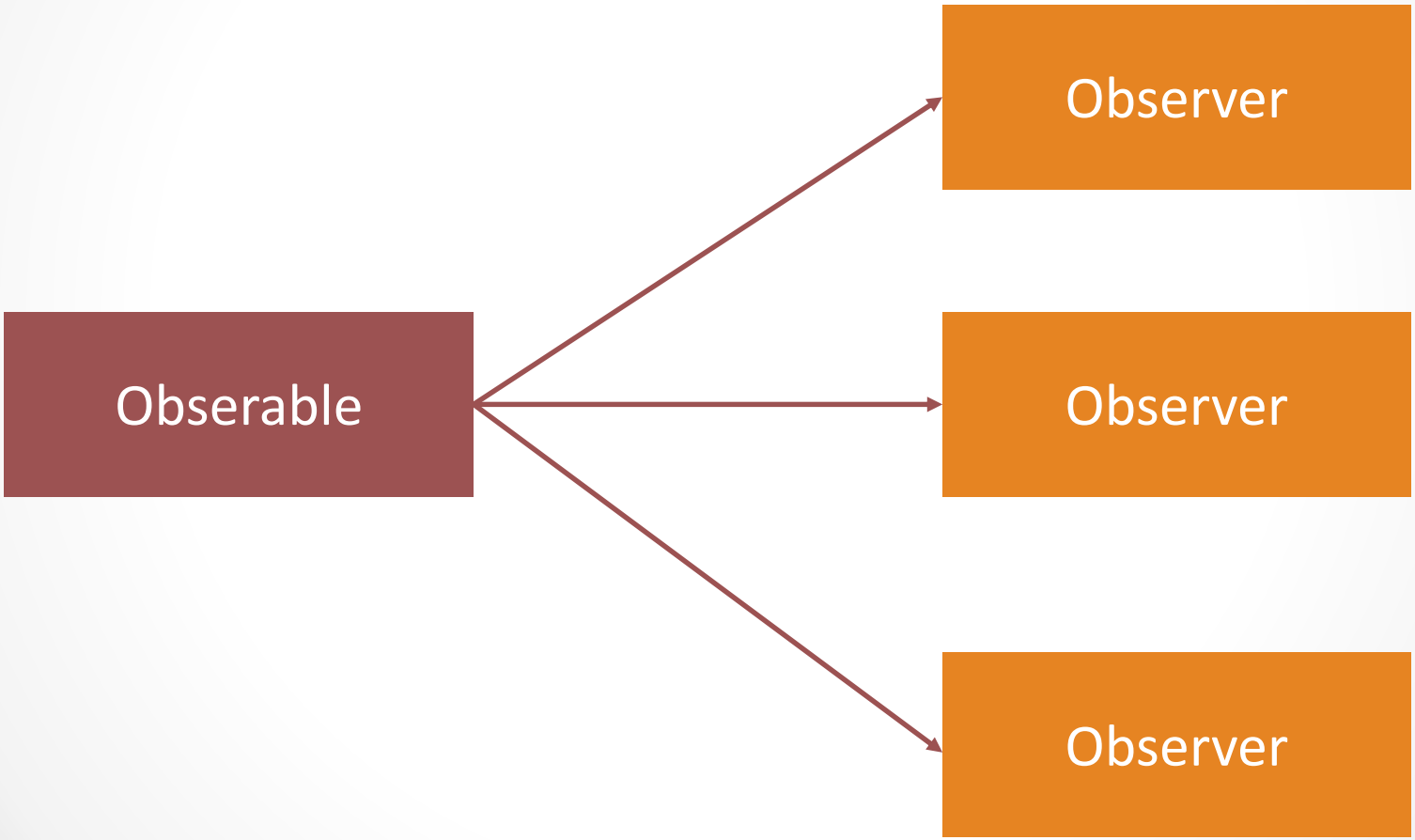
ReactiveX

- ✓ Best ideas from **Observer/Iterator** pattern and functional programming
- ✓ Push-model instead of pull-model
- ✓ Developed by **Netflix**
- ✓ Implemented in Java as **RxJS**
- ✓ Supports Java, Scala, Clojure, Swift and .NET

RxJS 6

- ✓ Push-based collection of events
- ✓ Implementation of observer design pattern
- ✓ Observable object sends notifications
- ✓ Observer object receive notifications

Reactive communication in RxJS



- Sergiy Morenets, 2020

Observable. Creation


```
let observable$: Observable<Number> = EMPTY;
observable$ = of(1, 2);
observable$ = range(1, 10);
observable$ = from([1, 2, 3]);

observable$ = throwError(new Error('Unexpected'));
```

import from ‘rxjs’

```
observable$.subscribe((text) => console.log(
  'Event received ' + text));
```

Similar to try/catch/finally

```
observable$.subscribe((text) => console.log(
  'Event received ' + text),
  (err) => console.log('Error ' + err),
  () => console.log('Process completed'));
```

- Sergiy Morenets, 2020

Observable. Operations

Enable tree-shaking

Creates chain of Observable operators

```
range(1, 10).pipe(map( project: (x: number) => x * 10),
  filter( predicate: (x: number) => x % 2 === 0))
  .subscribe(elem => console.log('Even number ' + elem));
```

Applies predicate function to each element of the stream

Task #20. RxJS

1. Create few Observable objects, using **from(), range()** and **of ()** functions.
2. Subscribe to observable to print received events or generated errors.
3. Create Observable that pushes **Latin** lowercase letters (from ‘a’ to ‘z’) and verify its behavior.

Working with HTTP

- Use XHR or JsonP approaches(**HttpModule** or JsonpModule)
- By default HTTP requests are made using XMLHttpRequest
- In Angular 2(and later) all the requests are based on **Observables**
- **Http service** has get/post/put/delete methods

Http service. Text files

Provided by HttpModule

```
export class HomeComponent {

  constructor(http: Http) {
    http.get( url: '/users.txt')
      .subscribe( next: response => {
          console.log('Status ' + response.statusText);
          console.log('Headers' + response.headers);
          console.log('Text ' + response.text());
        }
      );
  }
```

```
name=Sergey
sirname=Morenets
```

Observable

Function	Description
subscribe()	Apply function to each event(message)
map()	Convert event(perform transformation)
filter()	Filters event by some condition
reduce()	Convert stream into single value by consequently applying function to each pair of values
merge()	Merge two streams
take(N)	Limits stream to first N events
flatMap()	Converts stream of streams into single stream
range()	Create stream of sequential numbers

Http service. Sending request

```
constructor(private http: Http) {
  http.get( url: '/users.txt')
    .subscribe(response => console.log(
      'response ' + response.text()));
```

Deprecated since 4.3

```
export class HomeComponent {

  constructor(http: HttpClient) {
    http.get( url: '/users.json')
      .subscribe( next: response => {
          console.log('Data' + response);
        }
      );
  }
```

Introduced in 4.3
with HttpClientModule

HttpClient. Data binding


```
export class PanelHeaderComponent {
  users: User[];

  constructor(http: HttpClient) {
    http.get<User[]>('/user.json').subscribe(res => this.users = res);
  }

users$: Observable<User[]>;

constructor(http: HttpClient) {
  this.users$ = http.get<User[]>('/user.json');
}
```

Unsubscribes automatically and makes component stateless

```
<div>{{users$ | async}}</div>
```

HttpClient. Data binding


```
@Pipe({name: 'async', pure: false})
export class AsyncPipe implements OnDestroy, PipeTransform {
  private _latestValue: any = null;
  private _latestReturnedValue: any = null;

  private _subscription: SubscriptionLike|Promise<any>|null = null;
  private _obj: Observable<any>|Promise<any>|EventEmitter<any>|null = null;
  private _strategy: SubscriptionStrategy = null !;
```

```
ngOnDestroy(): void {
  if (this._subscription) {
    this._dispose();
  }
}
```

- Sergiy Morenets, 2020

HttpClient. Data binding


```
export class PanelHeaderComponent {
  user$: Observable<User>;
```

```
<div>
  <span>{{ (user$ | async)?.id}}</span>
  <span>{{ (user$ | async)?.name}}</span>
</div>
```

```
<div *ngIf="user$ | async as userModel">
  <span>{{userModel.id}}</span>
  <span>{{userModel.name}}</span>
</div>
```

HttpClient. Getting HTTP response


```
totalCount: string;

constructor(private http: HttpClient) {

  const resp: Observable<HttpResponse<User[]>> =
    this.http.get<User[]>('/users.json',
      {observe: 'response'})
      .pipe(catchError(this.handleError));

  resp.subscribe(res => this.totalCount =
    res.headers.get('x-total-count'));
```

Wrapper for HTTP response

Configure to return full response (doesn't need body)

Error handler

```
private handleError(err: HttpErrorResponse): Observable<any> {
  console.log(err.message);

  return throwError('Internal error');
}
```

Task #21. Working with Http

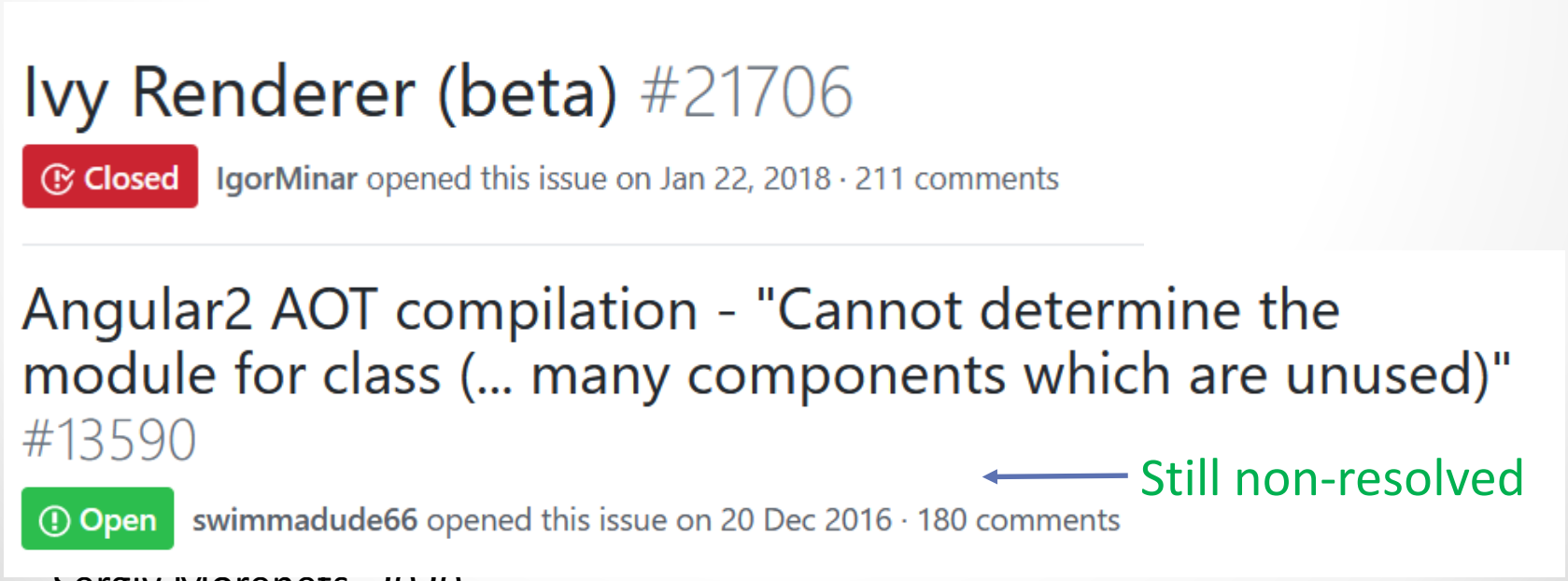
1. Create new **NodeJS** application in WebStorm. Review **app.js** file. Change the default port using line *app.set('port', 3000);*
2. Create JSON file **books.json** in the root folder that will contain list of book objects.
3. Replace '/users' with **'/books'** in app.js and use this line to send file content in **users.js**:
4. Run NodeJS application and verify that URL http://localhost:3000/books works

Performance improvements

- ✓ Bundle size (including tree-shaking)
- ✓ Rendering time
- ✓ Build speed
- ✓ Memory usage

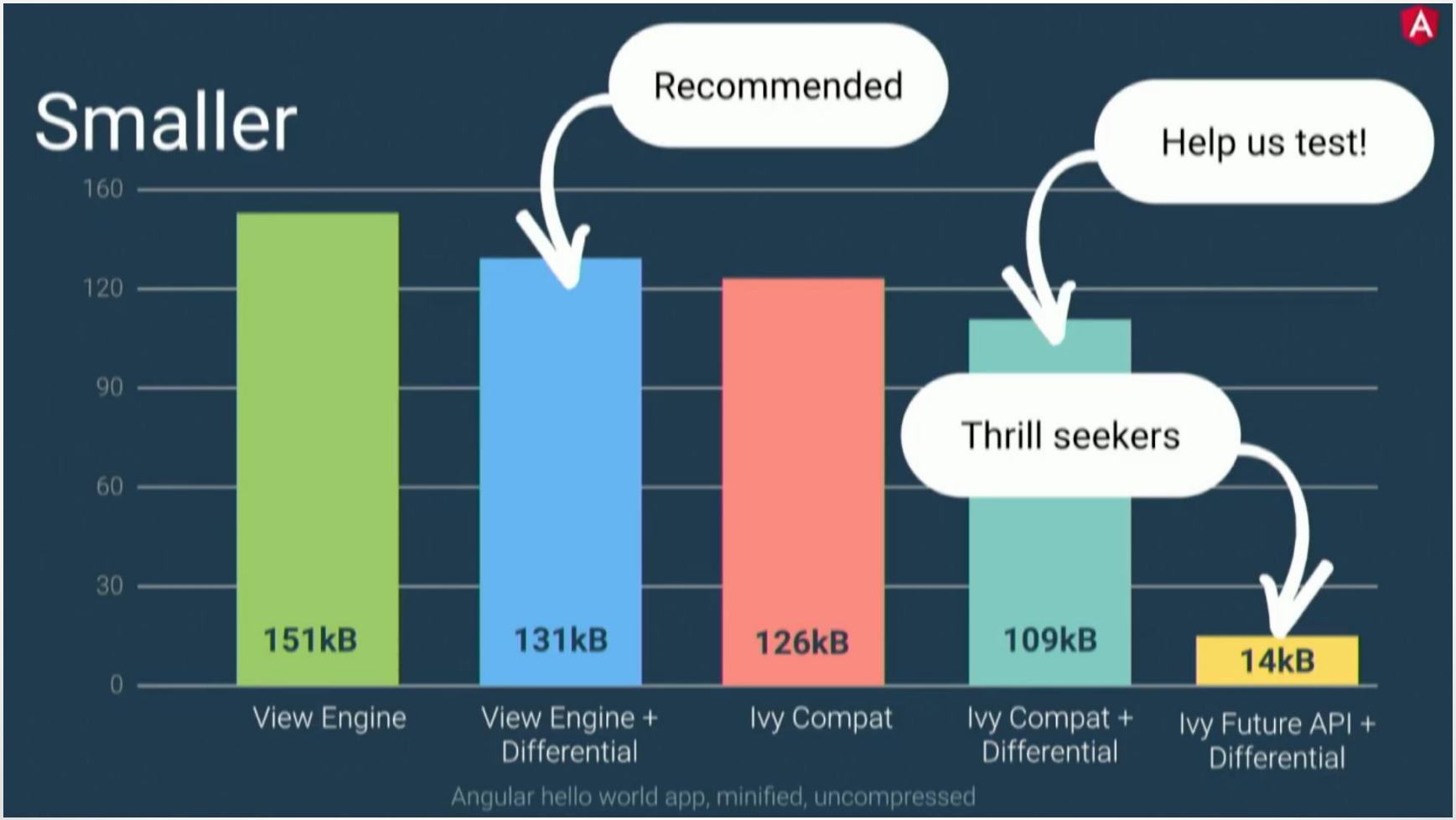
Ivy Renderer

- ✓ Compilation and rendering pipeline
- ✓ Generates code that easier to read and debug at run-time
- ✓ Faster re-build time
- ✓ Improved payload size and template type checking
- ✓ Backwards compatibility



- Sergiy Morenets, 2020

Ivy Renderer. Plans



- Sergiy Morenets, 2020

Rendering in Angular

- Angular framework contains compiler and runtime engine
- Compiler translates templates into JavaScript files with some transformations (removing type declarations, lowering declarations to ES5)
- Each compiled components contains two set of instruction (view creation and change detection)
- Runtime engine executes JavaScript code
- Angular (since v4.0) used ViewEngine rendering architecture
- ViewEngine has to be part of the bundle
- Limitations are absence of incremental compilation and no tree sharking in ViewEngine

Rendering in Angular

- ✓ Used Virtual DOM
- ✓ Virtual DOM creates new component tree when change detection happens (more memory footprint)

Ivy Renderer Architecture

- ✓ Moved from virtual DOM(used in React/Vue) to **incremental DOM**
- ✓ Incremental DOM enables **better** bundle size and memory footprint
- ✓ Each compiled component has built-in instructions to execute (not interpret) at run-time
- ✓ Angular decorators are compiled to class static properties
- ✓ Allows to bootstrap/render component **without modules**
- ✓ No intermediate wrapper classes (component.metadata.json and component.ngfactory.json)
- ✓ Compiled templates will be stored in the static fields of class improving overall tree shaking process

Ivy Renderer Architecture

- Libraries should use Angular **Package format**
- **ngcc** (Angular compatibility compiler) is a new compiler that convert and compile the libraries
- Libraries compatible with ViewEngine are converted into Ivy instructions
- Libraries should be created by View Engine Compiler (ngcc)
- Should have great impact on Angular Elements

Ivy. Lazy loading at component level


```
@Component(...)
export class AppComponent{
  constructor(
      private viewContainer: ViewContainer,
      private cfr: ComponentFactoryResolver) {

    // lazy click handler
    async lazyload() {
      // use the dynamic import
      const {LazyComponent} = await import('./lazy/lazy.component');
      this.viewContainer.createComponent(LazyComponent);
    }
  }
}
```

- Sergiy Morenets, 2020

Ivy Renderer. Example

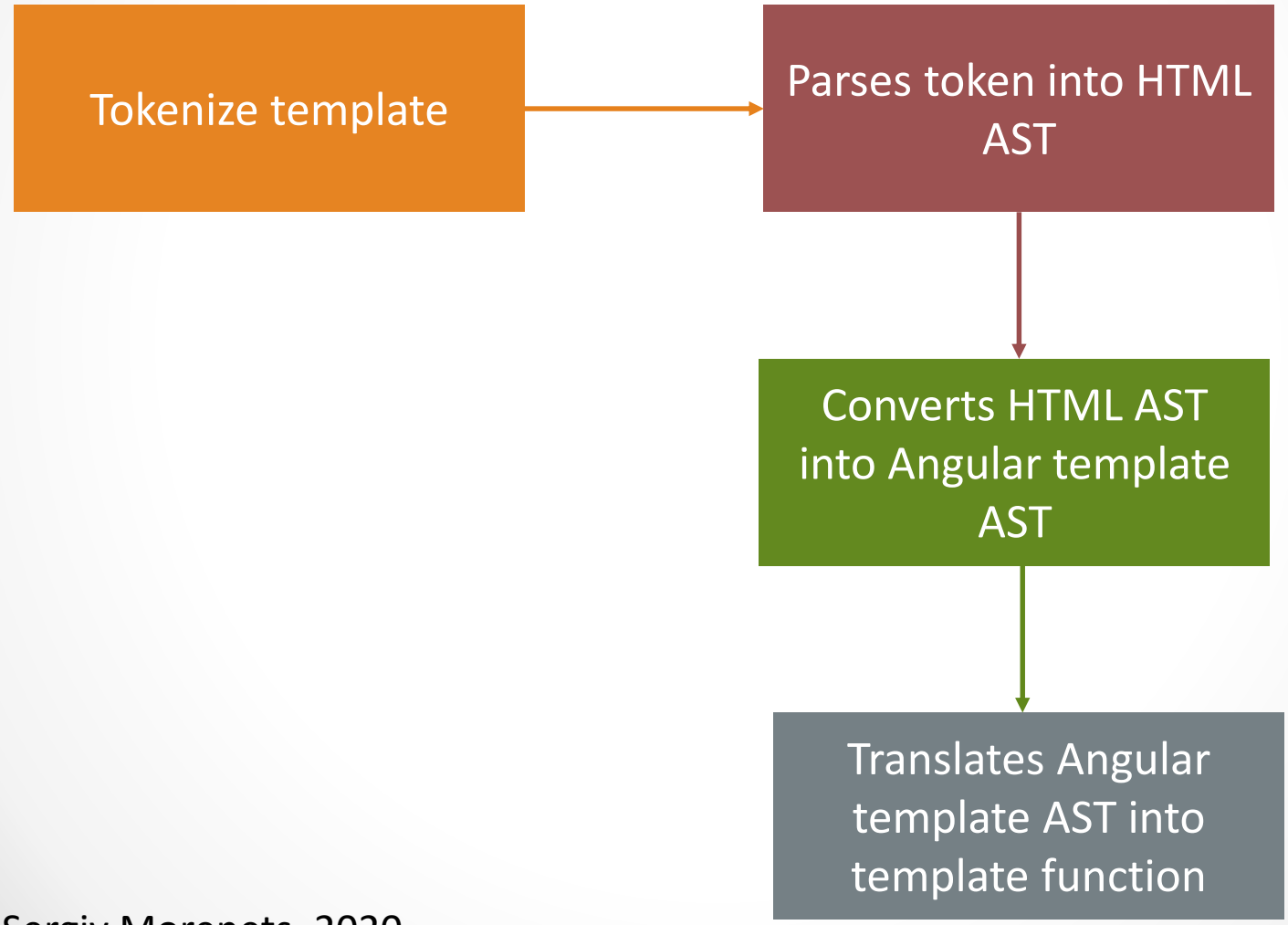

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <div style="text-align:center">
      <h1>
        Welcome to {{ title }}!
      </h1>
    </div>
  `,
  styleUrls: []
})
export class AppComponent {
  @Input() title = 'Angular!';
}
```

Ivy Renderer. Compiled code


```
export class AppComponent {
    constructor() {
        this.title = 'Angular';
    }
}
AppComponent.ɵfac = function AppComponent_Factory(t) { return new (t || AppComponent)(); };
AppComponent.ɵcmp = i0.ɵɵdefineComponent({ type: AppComponent, selectors: [["app-root"]],
  inputs: { title: "title" }, decls: 3, vars: 1,
  consts: [[2, "text-align", "center"]],
  template: function AppComponent_Template(rf, ctx) {
    if (rf & 1) {
        i0.ɵɵelementStart(0, "div", 0);
        i0.ɵɵelementStart(1, "h1");
        i0.ɵɵtext(2);
        i0.ɵɵelementEnd();
        i0.ɵɵelementEnd();
    } if (rf & 2) {
        i0.ɵɵadvance(2);
        i0.ɵɵtextInterpolate1(" Welcome to ", ctx.title, "! ");
    } }, encapsulation: 2 });
```

Compiling template



Ivy transformation


```
@Component({ /*...*/ })
class MyComponent { }
```

.ngfactory.js

```
class MyComponent {
  static ngComponentDef = defineComponent({ /*...*/ })
}
```

```
@Directive({ /*...*/ })
class MyDirective { }
```

```
class MyDirective {
  static ngDirectiveDef = defineDirective({ /*...*/ })
}
```

Lazy loading for components


```
@Component(...)
export class AppComponent{
  constructor(
      private viewContainer: ViewContainer,
      private cfr: ComponentFactoryResolver) {

    // lazy click handler
    async lazyload() {
      // use the dynamic import
      const {LazyComponent} = await import('./lazy/lazy.component');
      this.viewContainer.createComponent(LazyComponent);
    }
  }
}
```

- Sergiy Morenets, 2020

Ivy Renderer. Build size

enable-ivy flag produces larger bundle #13922

Open mohammedzamakhan opened this issue 11 days ago · 1 comment

```
Time: 11633ms
chunk {0} runtime.a5dd35324ddfd942bef1.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} es2015-polyfills.af54ab9c9df51e84c057.js (es2015-polyfills) 61.9 kB [initial] [rendered]
chunk {2} main.844e47fb990ff6ecca1c.js (main) 151 kB [initial] [rendered]
chunk {3} polyfills.407a467dedb63cfdd103.js (polyfills) 41 kB [initial] [rendered]
chunk {4} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
```

Angular 7

```
Time: 117121ms
chunk {0} runtime.a5dd35324ddfd942bef1.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} es2015-polyfills.af54ab9c9df51e84c057.js (es2015-polyfills) 61.9 kB [initial] [rendered]
chunk {2} main.8ac9417e5814bfdedf12.js (main) 22.2 kB [initial] [rendered]
chunk {3} polyfills.407a467dedb63cfdd103.js (polyfills) 41 kB [initial] [rendered]
chunk {4} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
```

Angular 8 with Ivy

- Sergiy Morenets, 2020

Ivy Renderer. Build size

Angular 7.2 ES5

- vendor.js **192.88 kb**
- main.js **80.98 kb**
- polyfills.js 35.18 kb

Angular 7.2 ES2015

- vendor.js **184.81 kb**
- main.js **76.13 kb**
- polyfills.js 21.64 kb

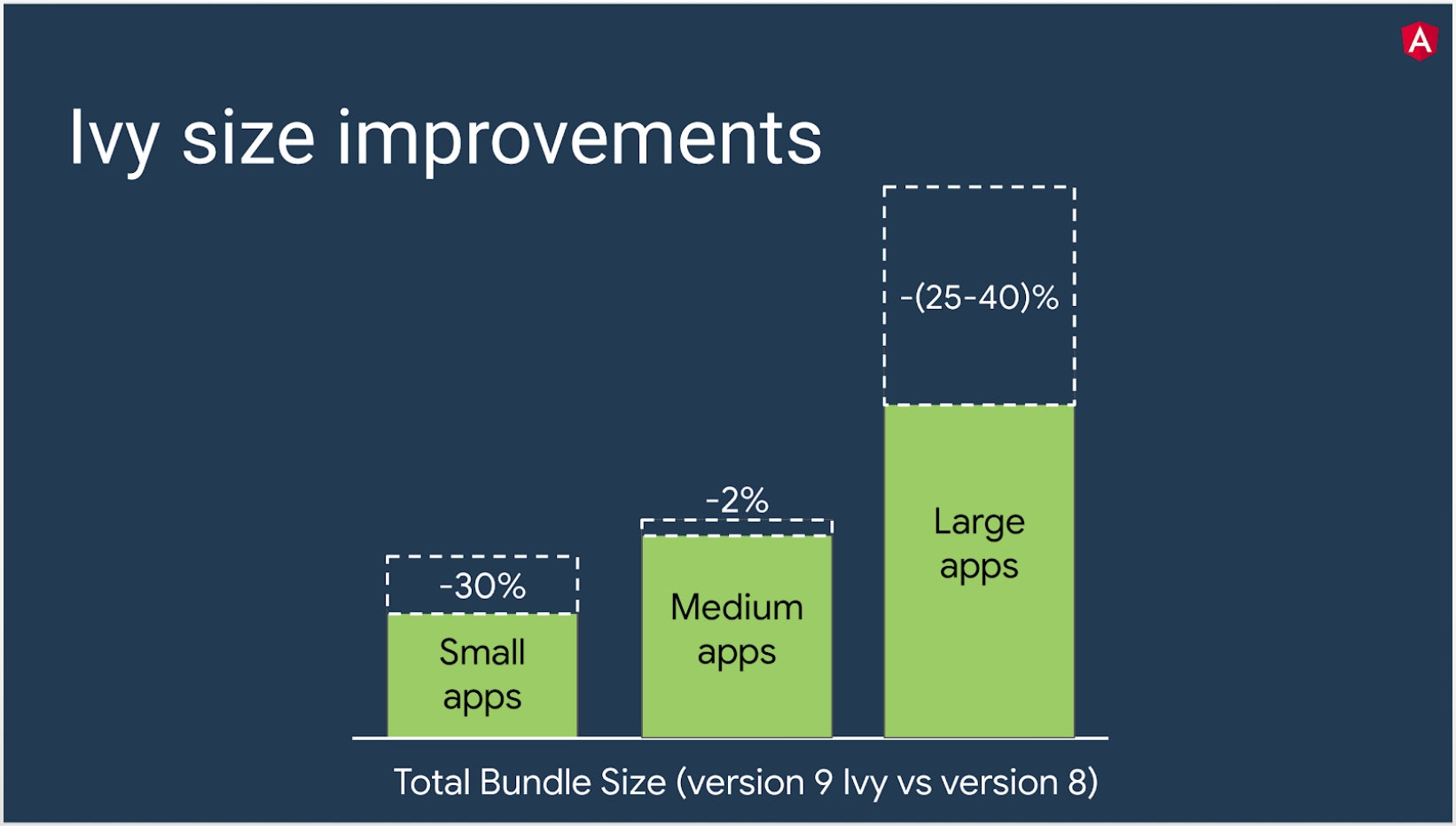
Angular 9.0-next.11 ES5

- vendor.js **234.35 kb**
- main.js **77.04 kb**
- polyfills.js 43.24 kb

Angular 9.0-next.11 ES2015

- vendor.js **223.46 kb**
- main.js **71.7 kb**
- polyfills.js 14.37 kb

Ivy Renderer. Build size



- Sergiy Morenets, 2020

Task #22. Ivy renderer

1. Build your project: *ng build --prod* and remember bundle size and build time.
2. Add this block to **tsconfig.app.json**:

```
"angularCompilerOptions": {
   "enableIvy": false
}
```

3. Build the project again: *ng build --prod*. What is build time and bundle size now?

Summary

- ✓ If you are back-end developer then you would love Angular for static typing/OOP
- ✓ If you are front-end developer then Angular is perfect chance to learn & improve design/architecture skills

Angular is created for enterprise-scale applications

Style guide

- ✓ Project structure conventions
- ✓ File names conventions
- ✓ Code conventions

Style guide. Project structure

- ✓ Put bootstrapping and platform logic, error handling into **main.ts**
- ✓ Project source code should be in **src** folder and vendor code outside of **src** folder
- ✓ Each feature module should be in the separate folder(for example, src/app/messages)
- ✓ Use maximum 7 files in the folder
- ✓ Extract template with more than 3 lines into separate file

Style guide. File naming

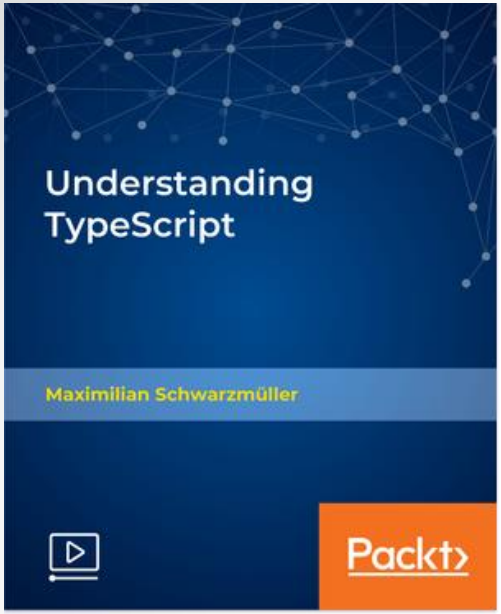
- ✓ Use **feature.type.ts** template for the project files
- ✓ Include .module, .component, .service words in the file names
- ✓ Use dash to separate words in the name
- ✓ Use dot to separate name and type
- ✓ Specify unit test file name as main file name and “spec” suffix

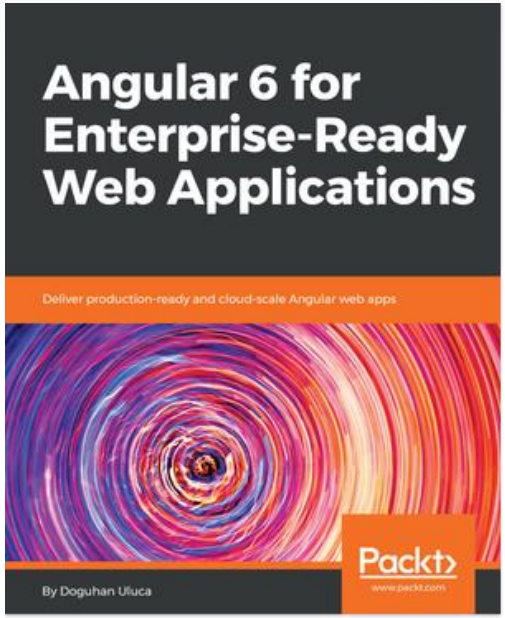
Style guide. Source code

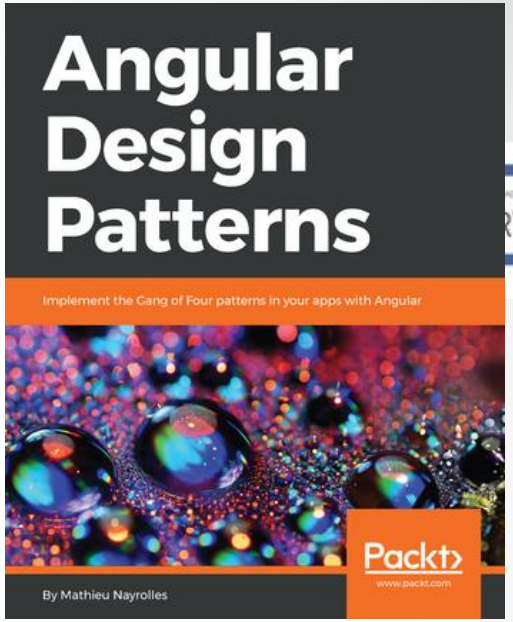
- ✓ Use SRP everywhere
- ✓ Limit files up to 400 lines of code
- ✓ Limit function up to 75 lines
- ✓ Use **upper** CamelCase for class/interface names
- ✓ Use **lower** CamelCase for property/method names
- ✓ Include Component/Service/Module into naming
- ✓ Use lower **kebab-case** for selector naming
- ✓ Use application/feature prefixes for selectors
- ✓ Use **const** for immutable variables
- ✓ Avoid underscore for private properties

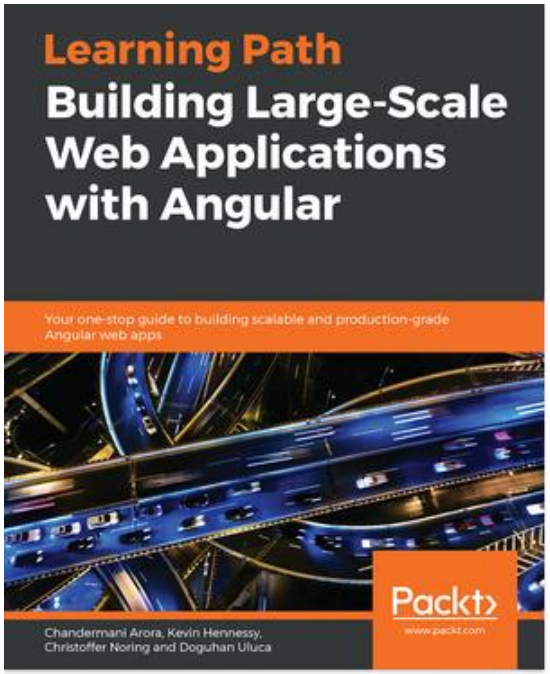
Style guide. Source code

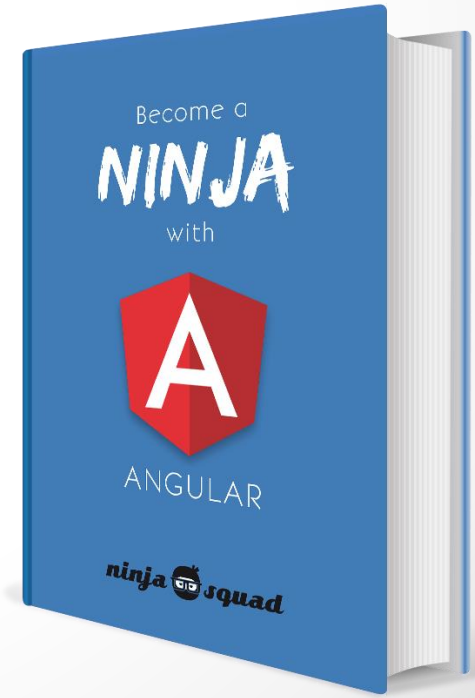
- ✓ Use @Input/@Output instead of inputs/outputs attributes
- ✓ Put properties before the methods
- ✓ Put public member before private ones
- ✓ Delegate reusable logic to the services
- ✓ Put presentation logic into component, not template
- ✓ Use directives when you need presentation logic without template
- ✓ Use singleton services with shared data/behavior
- ✓ Use services for HTTP operations, local storage or another data operations











- Sergiy Morenets, 2020

✓ Sergiy Morenets, sergey.morenets@gmail.com