HOW

Hands On Workshop

By : Vijay Shivakumar

What do we need ?

Technical Skill : HTML5, CSS3, JavaScript 1.8.5

Hardware and software

IDE : visual studio code

Browsers : chrome latest

Platform : nodejs latest

Database : mongodb / mlab

Version Control : git

Network : internet access to download from git and npmjs.org

Objective

About this course

Understand and explore ES6 / ES7

Write Programs using Typescript 3x

Understand members of Angular bundle

Develop programs using Angular platform

Workflow for fast Angular application creation with Angular CLI

Unit Testing Angular code

About | me

Vijay Shivakumar

Designer | Developer | Trainer

Adobe | CERTIFIED EXPERT Flex® with AIR

Training & Consultation of
Contemporary Web Technologies and Adobe products from past 14 years

About | you

Designer

Developer

Architect

Business Analyst

Technology Enthusiast

Benefits

Angular Advantage

- Open Source
- Reduction in development time
- Latest Compliances
  - ES6
  - Modular
  - Internationalization and Accessibility
- Performance
- Popularity / Availability of resources
- Clear Documentation

Angular | Features

Leverages on new HTML5 Features
Includes cutting edge JavaScript features ES6, ES7
TypeScript for strong data typing
Better error handling
Speed and performance
Modular approach
Hybrid (Mobile, Tablet and Web support)
Feature rich to create SPAs
(DOM handling, 2 way Binding, Routing, Animation, Validation, Ajax, consumes RESTful APIs)

Setup / Tooling

What to know / use ?

NodeJS
TypeScript
TraceurJS
BableJS
SystemJS
Webpack
Express
Jasmine
Karma

Git
MongoDB Mlab ID / Firebase ID
NodeMon
NVM

Angular CLI

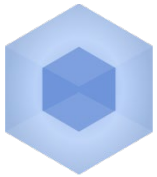
What is used...

NodeJS
TypeScript
TraceurJS
BableJS
SystemJS
Webpack
Express
Jasmine
Karma

Git
MongoDB Mlab ID / Firebase ID
NodeMon
NVM

Build Tools

Using Angular-CLI



ES5	None
ES6	Traceur BableJS SystemJS Webpack Express
TypeScript	TypeScript SystemJS Traceur / BableJS Webpack Express

Architecture

What make Angular ?

One way data flow : Data flow only from parent to child unlike angular1

Dependency Injection : Resources are passed to a component as and when required

Components : Creates a custom tag where the component replaces its content

Directives : Adds new functionality to HTML elements that they never had

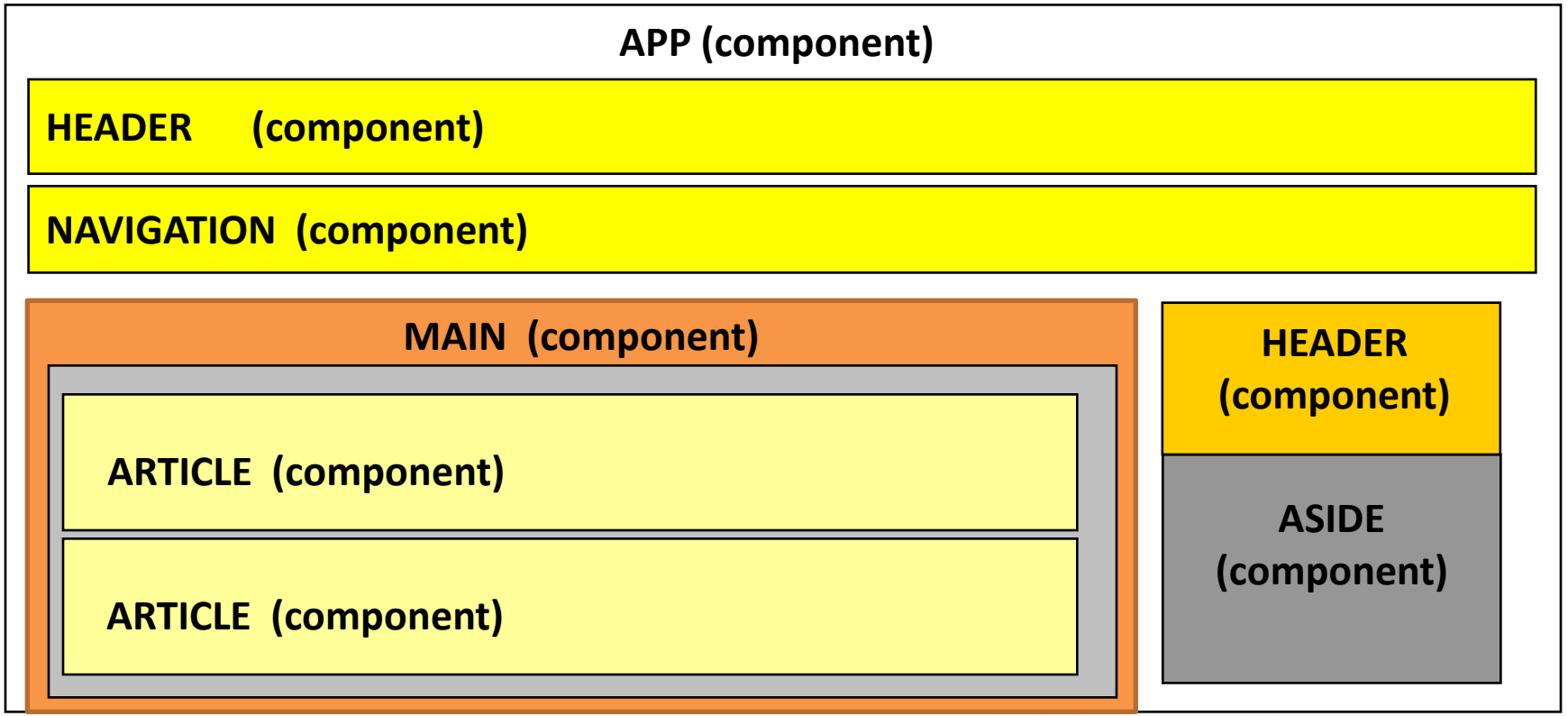
Templates : Are separate from component

Zone.js : Manages change detection

Rendering Targets : Render an app for many devices with

Browser-platform

Browser-platform-dymanic


APP (component)
HEADER (component)
NAVIGATION (component)
MAIN (component)
ARTICLE (component)
ARTICLE (component)
HEADER (component)
ASIDE (component)

Features Of Angular

Module
Component
Services
Pipe
Directive
Routes

Module

What is it doing..?

Is different from ES6 module

Every application must have at least 1 module (root module)

Root module is decorated with 'NgModule' from @angular/core

import : [FirstModule, SecondModule],

declarations : [Components, Pipes, Directives]

providers: [servicesToInject1, servicesToInject2]

bootstrap : [mainComponent]

Component | What is it ?

A basic Component has two parts.

1. A Component decorator
2. A component definition class

Component Decorator :

We can think of decorators as metadata added to our code.
When we use @Component on a class,
we are “decorating” that class as a Component.

Component Class :

Will have a constructor, properties , methods and life cycle events by default

Component

What is it doing..?

A component is a combination of a view (the template) and some logic

Is decorated with 'Component' from @angular/core

By convention every application must have a main component which is bootstrapped via module.

selector: 'aDomElement'(usually a custom tag name)

template:

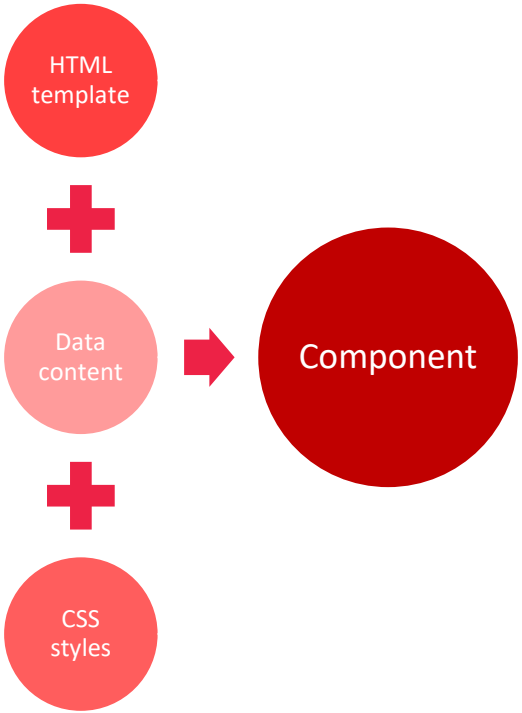
templateUrl:

styles:[]

stylesUrl:[]

Component

What is it ?



Templates | View

template : Inline

templateUrl : external

Display Data

Format Data

User Interaction

Avoid business logic

Styling | Component v/s Page

Styling a component

```
styles:[`
        selector { property : value; property : value }
        selector { property : value; property : value }
        `]
```

OR

```
stylesUrl:[ "location/style.css", "anotherlocation/style.css" ]
```

Styling a page

That you define in the html page in the head or body section

Binding

Property Binding
- interpolation / innerHTML
- value binding
- attribute binding
- style binding

Event Binding

One way Binding

Two way Binding

Binding Interpolation

Property Binding

```
<h1>{{ 2+2 }}</h1>
<h1>{{ user.name }}</h1>
<h1 [textContent]="user.name"> </h1>
<h1 bind-innerHTML='user.name'></h1>
<h1>{{  user.message()  }}</h1>
```

Keep it simple and fast (they should not take more time to compute)

Avoid multiple statements

You can not use assignment operators e.g. = , +=, ++, - - in property binding

You can not create an object e.g. new User().

Binding Interpolation

Property Binding

```
<h1>{{ 2+2 }}</h1>
<h1>{{ user.name }}</h1>
<h1 [textContent]="user.name"></h1>
<h1 bind-innerHTML='user.name'></h1>
<h1>{{  user.message()  }}</h1>
```

Tip: Use ? to handle undefined
Use the safe navigation operator
e.g. <h1>{{ no-user?.prop }}</h1>

Keep it simple and fast (they should not take more time to compute)

Avoid multiple statements

You can not use assignment operators e.g. = , +=, ++, -- in property binding

You can not create an object e.g. new User().

Event Binding

Statement Binding

```
<button (click)="callfun()"> Click Me </button>
<input (keydown.space)="onSpaceBarDownEvent()"> Click Me </button>
<button on-click="onButtonClick()">Click me!</button>
```

Keep it simple and fast (they should not take more time to compute)

The callback functions can take a single parameter which is referred as $event

(If you wish to send multiple params you can wrap them in an object and send)

Avoid business logic on templates

You can not create an object e.g. new User().

Events

Element events supported

mouseenter	dragover	cut
mousedown	drop	copy
mouseup	focus	paste
click	blur	keydown
dblclick	submit	keypress
drag	scroll	keyup

Style Binding | class / ngClass

[class.className] =“stronghero”

[ngClass]="['box', 'boxer']"

[ngClass]={ expression that returns a string, object or an array}

In the example below both stronghero and boxclass can be applied if it matches the conditions

Eg;

[ngClass]="{ stronghero: heroPower > 5, boxclass: rating > 0.5}

Tip: conditionally applied classes will append to existing classes

Style Binding

style / ngStyle

As a property binding

[style.color] =“ ‘#333’ ” or
[style.background-color] =“ ‘yellow’ ”

[ngStyle]={ expression that returns a style value}

In the example below the ternary operator will return a style property and value combination

Eg;

[ngStyle]="{‘color’: ‘red’, ‘background-color’ : ‘gray’}”

[ngStyle]="{‘color’: heroPower > 5 ? ‘green’ : ‘red’}”

Directive

Structural Directives

*ngIf
*ngFor
*ngSwitch

NgIf	*ngIf
NgSwitch	*ngSwitch
NgFor	*ngFor
NgStyle	[ngStyle]
NgClass	[ngClass]
NgNonBindable	ngNonBindable

Structural Directive

Built in Directives

Adds or removes the DOM contents or they change the structure of DOM
hence the name

```
<ul>
    <li *ngFor = “let user of users”> </li>
</ul>
```

In this example it shall loop over and for users

and create a temp variable user in the scope of the loop

For every loop the li and its content is repeated

<li *ngIf=“true/false”>{{ data }} OR <li [hidden]=“true/false”>{{ data }}

Structural Directive

Built in Directives

Adds or removes the DOM contents or they change the structure of DOM
hence the name

```
<ul>
    <li *ngFor = “let user of users”> </li>
</ul>
```

In this example it shall loop over and for users

and create a temp variable user in the scope of the loop

For every loop the li and its content is repeated

<li *ngIf=“true/false”>{{ data }} OR <li [hidden]=“true/false”>{{ data }}

Tip: Usage of hidden is efficient

Structural Directive

The ng-template way

*ngIf
*ngFor
*ngSwitch

NgIf
NgSwitch
NgStyle
NgClass
NgFor
NgNonBindable

Structural Directive | Switch

```
<div [ngSwitch]="hero.power">
        <h1 *ngSwitchCase="8">Strong</h1>
        <h1 *ngSwitchCase="7">Weak</h1>
        <h1 *ngSwitchDefault>Recovering</h1>
</div>
```

Binding Summary

When to use what ?

{{}} for interpolation

[] for property binding

() for event binding

[(ngModel)] for two way binding

for variable declaration

* for structural directives

Pipes / Formaters

Built-in Pipes
Passing parameters to pipes
Chaining pipes
Async pipe
Custom pipes

Pipes

Format your output

DatePipe	used to format the date as needed
LowerCasePipe	converts to lowercase
UpperCasePipe	converts to uppercase
CurrencyPipe	applies a currency symbol and manage integer and decimals
DecimalPipe	manages decimal values
AsyncPipe	will deal with observable values and display latest result

Component Features

Local Variables
Style Behavior
Accessing Content
Component Communication
Life Cycle Hooks / Events

Local Variables

Temporary variables

```
<input type="text" #name>
{{ name.value }}
Or
<input type="text" #name>
<button (click)="name.focus()">Focus the input</button>
Or
<video #player src='myvideo.mp4'></video>
<button (click)="player.play()">Play </button>
Or  (usage of ref- attribute to create a local variable )
<input type="text" ref-name>
<button on-click="name.focus()">Focus the input</button>
```

Communication

Input & Outputs

@Input decorator from @angular/core

allows data inlet in to the component

@Output decorator from @angular/core

allows data to be sent from the component

Tip : only events are allowed to be used as an output

The same can be done with these properties of a component

input :

output :

You can use template variables to communicate from a child component to parent component

Communication | alias

@Input("externalName") internalName: string ;

Life Cycle Hook

Angular Events

ngOnChange : Happens when the component is changed via an input directive the event captured will have the previous value, currentValue and if the value is changed for first time.

Object { previousValue: undefined, currentValue: 0, firstChange: true }

ngOnInit : Only once when the component is initialized

ngOnDestroy : is called when the component is removed. Really useful to do some cleanup.

Life Cycle Hook

Angular Events

ngOnChanges : Happens when the component is changed via an input directive the event captured will have the previous value, currentValue and if the value is changed for first time.

Object { previousValue: undefined, currentValue: 0, firstChange: true }

ngDoCheck : happens when ever the component is changed via an input directive

ngOnInit : Only once when the component is initialized

ngOnDestroy : is called when the component is removed. Really useful to do some cleanup.

Sharing Data / Logic

Services
Dependency Injection

Services

Dealing with Data

- Reusable functionality shared across components yet independent from components (not tied to any specific component)
- Responsible for a single piece of functionality simple classes that fetch data or logic across components
- Deliver data or logic when and where it is needed yet encapsulates external interactions such as with data

Services | How to Create ?

1. Build a service
2. Register the service
3. Inject the service

Dependency Injection

Inversion of control

Dependency injection is a well-known design pattern.

A component may need some features offered by other parts of our app such as a services. (referred as dependency)

Instead of letting the component create its dependencies, the idea is to let the framework create them, and provide them to the component.

That is known as "inversion of control".

Declare dependencies with providers : [] either on module or on component

To inform Angular that this service has some dependencies itself,

we need to add a class decorator: @Injectable()

Dependency Injection | Example

```
class Heroes{
}

class Movies{
}
```

```
import { Heroes, Movies } from "./"
class HeroApp{

  heroes;
  movies:

   constructor(){
     this.heroes = new Heroes();
     this.movies = new Movies();
    }

}
```

Forms

Template Driven Forms
Data Driven / Reactive Forms
Form and Input state management
Validation

Template Forms

Easy to use

Similar to legacy forms

Uses 2 way data binding

Minimal component code

Automatically track form input element and state

Reactive Forms

More Flexible

Used for complex scenarios

Model is immutable

Easier to perform action upon value change

Reactive Transformations (debounce)

Add input elements dynamically

Unit test forms

Validation

Forms & User Inputs

html5 validations	css validations	JavaScript validations
minimum	.ng-invalid	input.invalid
maximum	.ng-touched	input.touched
required	.ng-valid	input.valid
pattern		

Forms are everywhere in an application...

FormControls

encapsulate the inputs in our forms and give us objects to work with them

Validators

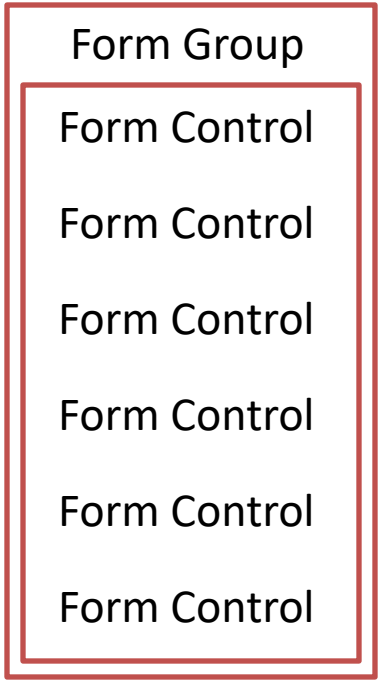
give us the ability to validate inputs, any way we'd like

Observers

let us watch our form for changes and respond accordingly

Validation | States

- Value Changed
  - Pristine
  - Dirty
- Validity
  - Valid
  - Invalid
- Visited
  - Touched
  - Untouched



FormControl

Represents a single input field - it is the smallest unit of an Angular form.

Eg:

```
// create a new FormControl with the value "foo"
let nameControl = new FormControl("foo")
let name = nameControl.value; // foo
// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty // -> false
nameControl.valid // -> true
<input type="text" [formControl]="name" />
```

FormGroup

Provides a wrapper interface around more than one FormControls so we can manage multiple fields to validate them.

Eg:

```
// create a new FormControl with the value "foo"
 let heroInfo = new FormGroup({
firstName: new FormControl("Bruce"),
lastName: new FormControl("Wayne"),
power: new FormControl("7")
})
```

NgRx / RxJS

Stores
State
Actions
Reducers
Effects

Why do we need it ?

Not for every project

Manage state

- if user is logged in or not...
- to keep data between 2 lists or component in sync
- make the application remember data when they move between routes

Redux / Rxjs | Principles

A Store : Single source of truth called store

A State : Read only and can be changed only by actions

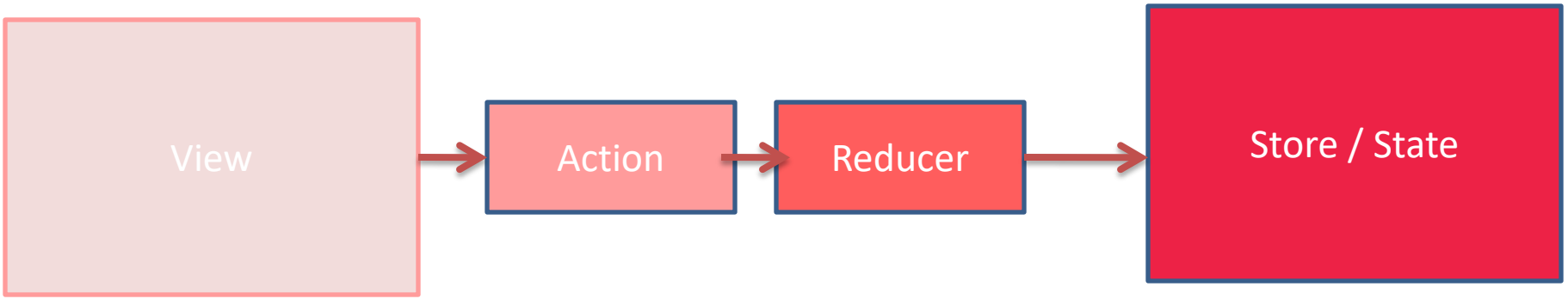
Reducers : Pure functions that update state

Rxjs | Principles

A Store : Single source of truth called store

An Action : State is read only and only changed by dispatching actions

Reducers : Changes are made using pure functions which are called reducers



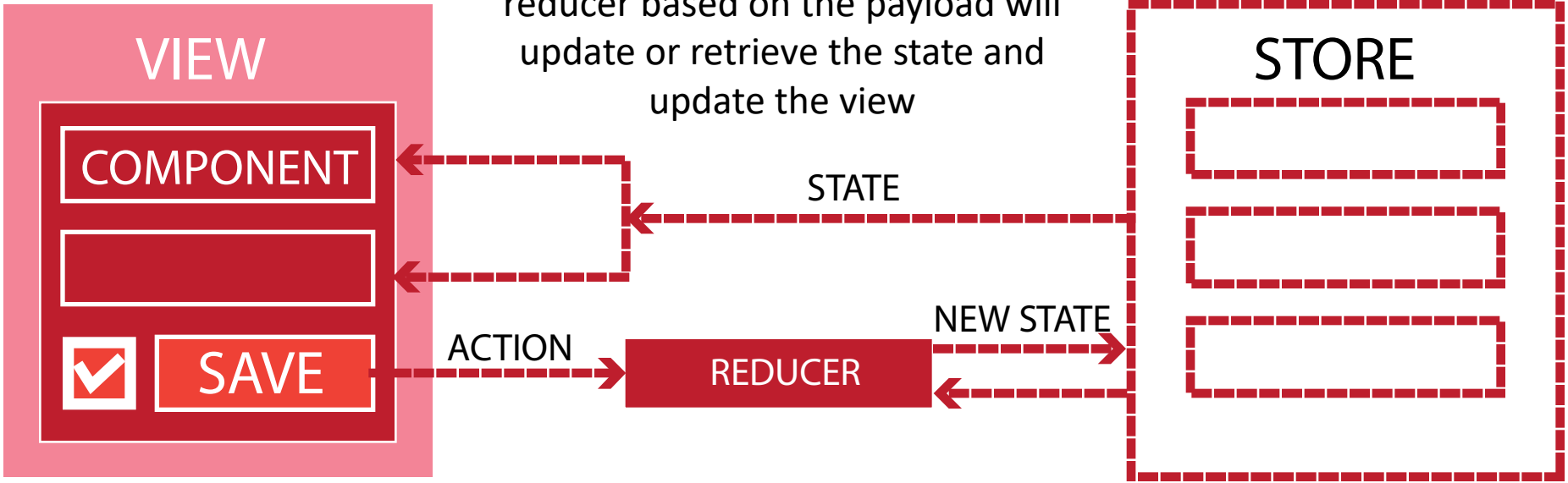
Store

What is it?

A Store : is a JavaScript object, like client side database

NgRx | Flow

View has a component which triggers an action with a payload to reducer, reducer based on the payload will update or retrieve the state and update the view



Routing

Navigation
Usage of History API
Passing and accessing data via parameters
Route Guards
Lazy loading

Routing

Navigating / Multipage

Routing : Map a URL to a state of the application
Angular supports HTML5 and Hash based URL Routing

Define Base Path : implemented by default with angular-cli <base href="/" />
recommended to be the first child of head tag

Add Module : in module import { RouterModule } from '@angular/router';

Import Router : import { Routes } from '@angular/router';

Configure Routes : export let ROUTES: Routes = [
{ path: '', component: HomeComponent },
{ path: about', component: AboutComponent }
];

Place Templates : <router-outlet>

Activate Routes : navigate to that path in browser url

Routing

Work flow

Users click on a link (created by routerlink)

Angular navigates to that link changing the location URL in the address bar

The location change triggers the router from the route configuration

Angular then loads the appropriate component in the template <router-outlet>

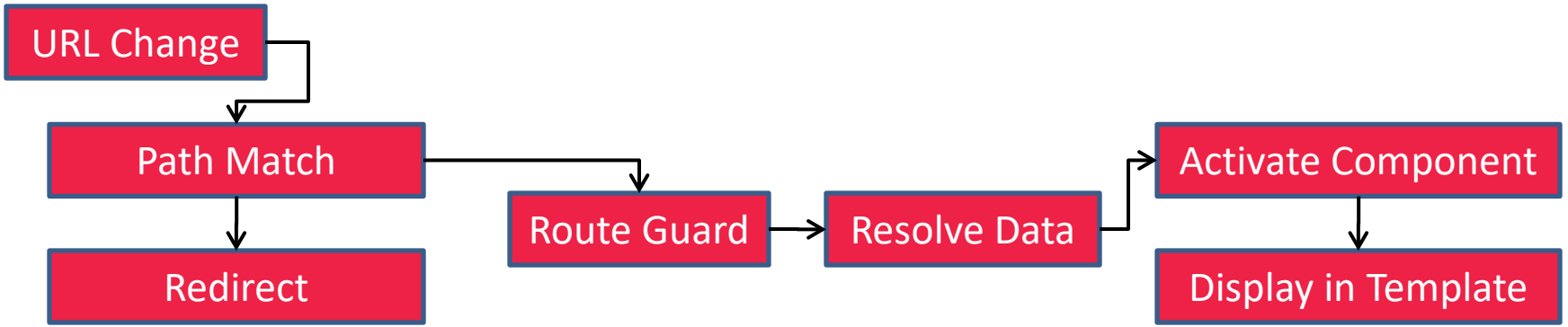
Routing | Work flow

Users click on a link (created by routerlink)

Angular navigates to that link changing the location URL in the address bar

The location change triggers the router from the route configuration

Angular then loads the appropriate component in the template <router-outlet>



Routing

Navigating / Multipage

redirectTo can be full or prefix

prefix is the default

full means the full path should resolve to any matching routing expression

route paths are case sensitive

route order does matter (components occupy the first matched path)

specific paths should be before less specific paths

e.g. wildcards should come last

redirects can not be chained i.e. one redirect can not match a path that does redirect again... (that won't work)

Routing | Passing Parameters

parameters

One component may have to pass data to another component in the next view

Pass simple data like ID's or Keywords

Optional Parameters

Useful to pass several arguments from one component to another without affecting the route configuration

QueryParameters

Useful to pass several arguments, allows us to preserve the data

Route Resolvers

Fetch data ahead of time for the component to create it.

Routing

Featured Modules

Using features are better to organize the routes

Imported featured routes are loaded first in order

Routing | Guards

Route Guards are services

Flow process for route guard

- CanDeactivate
- CanLoad
- CanActivateChild
- CanActivate : return can be boolean or data
- Resolve

Unit Testing | Fundamentals

Jasmine for assertion

Karma for testing angular modules

Angular CLI

Adding Libraries
Generating Parts of Angular Application
Building
Running
Testing
Serving in a browser

Thank you

vijay.shivu@gmail.com