

ANGULAR WORKSHOP

Naveen Pete

Agenda

- What are Frameworks?
- What is Angular?
- Angular Versions
- Why Angular?
- Where does Angular fit?
- Angular CLI
- TypeScript
- Angular Building Blocks
- Setting up Dev Environment
- Components
- Decorators
- Modules
- Data Binding
- Directives
- Pipes
- Component Interaction
- Services
- Routing
- Observables
- Forms
- Server Communication

What are Frameworks?

- Software Library
 - Collection of functions
 - Has well-defined interface
 - Reuse of behavior
 - Modular
- Software Framework
 - Provides
 - generic functionality
 - you the ability to customize the functionality according to your app needs
 - reusable environment
 - broad generic structure for your app

What are Frameworks?

- Library vs Framework
 - Library
 - Your code is in charge
 - Calls into the library when necessary
 - Framework
 - Framework is in charge
 - Calls into your code when needed
- Hollywood Principle
 - Do not call us, we will call you
- Inversion of Control

What is Angular?

- Developed in 2009 by Misko Hevery
- Currently maintained by Google
- Framework for building front-end JavaScript applications
- Angular apps
 - Can run on desktop and mobile devices
 - Are generally SPAs
- Open-source, TypeScript-based framework
- 'A' of MEAN stack

Angular Versions

- AngularJS (v1.x)
 - Aims to simplify the development and testing of web apps
 - Worked on the concept of scope and controllers
 - Initial release, v0.9.0 – Oct 2010
 - Latest release, v1.6.9 – Feb 2018
- Angular 2
 - Added component as a key building block
 - Complete re-write of AngularJS, no backward compatibility
 - Released in Sep 2016

Angular Versions

- Angular 4
 - Apps are smaller & faster
 - AOT compilation, Angular Universal - SSR
 - Backward compatible with Angular 2
 - Released in Mar 2017
- Angular 5
 - Smaller, faster and easier to use
 - Build optimizer, compiler improvements
 - New HttpClient, pipes, router lifecycle events
 - Released in Nov 2017

Why Angular?

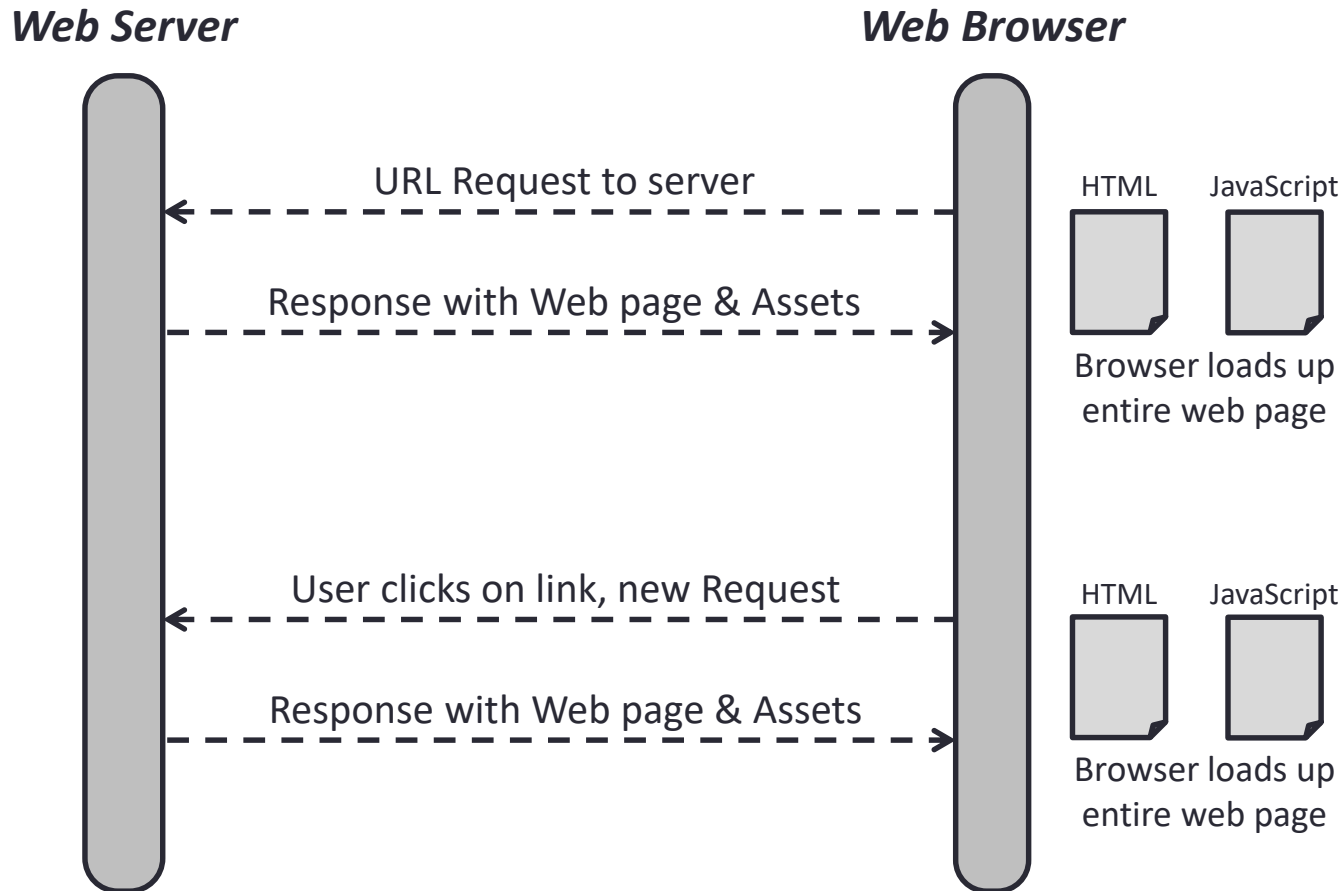
- Single Page Apps (SPA)
 - Better user experience
 - Reduced full page reloads
 - Better overall performance
 - Less network bandwidth
- Proven software patterns and practices
 - Model View Controller (MVC)
 - Model View ViewModel (MVVM)
 - Dependency Injection (DI)
- Declarative programming
 - Better readability, concise code
 - Better developer productivity
 - Faster development

Why Angular?

- Component based
 - Reusable
- Structures app code
 - Modular, Maintainable, Scalable
- Cross platform, mobile support
 - Target multiple browsers, platforms & devices
- Decouples DOM manipulation from app logic
 - Testable, TDD
- Move app code forward in the stack
 - Reduces server load, reduces cost
 - Crowd sourcing of computational power

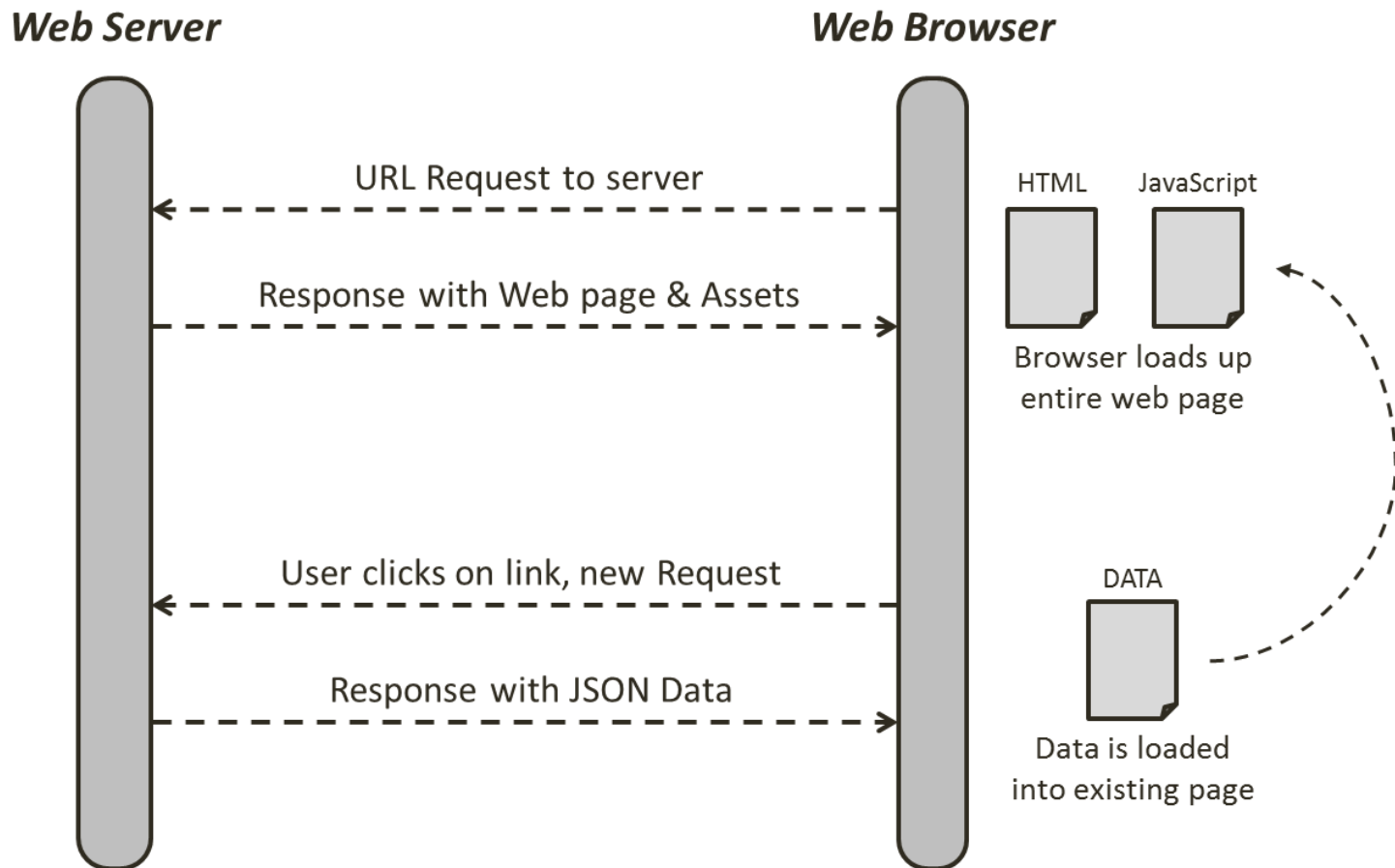
Traditional Web App

Request & Response

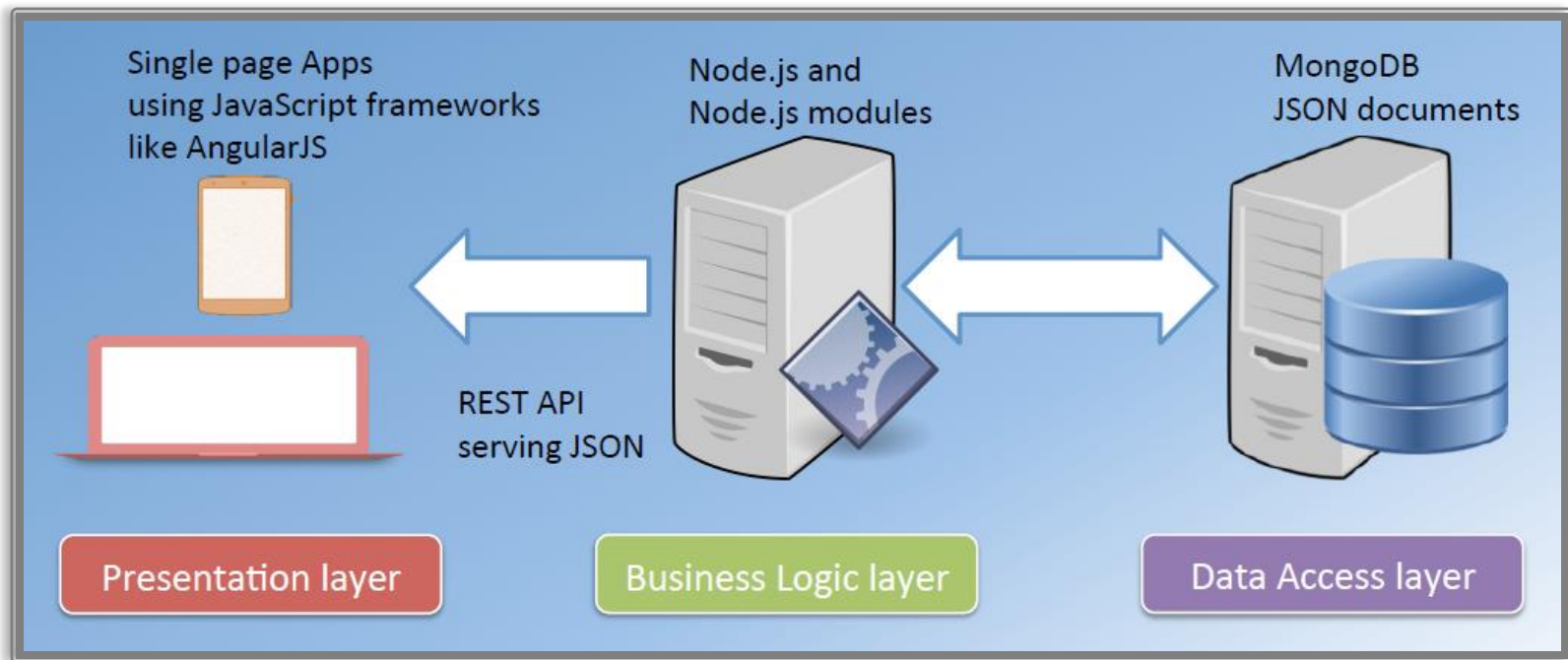


Angular App

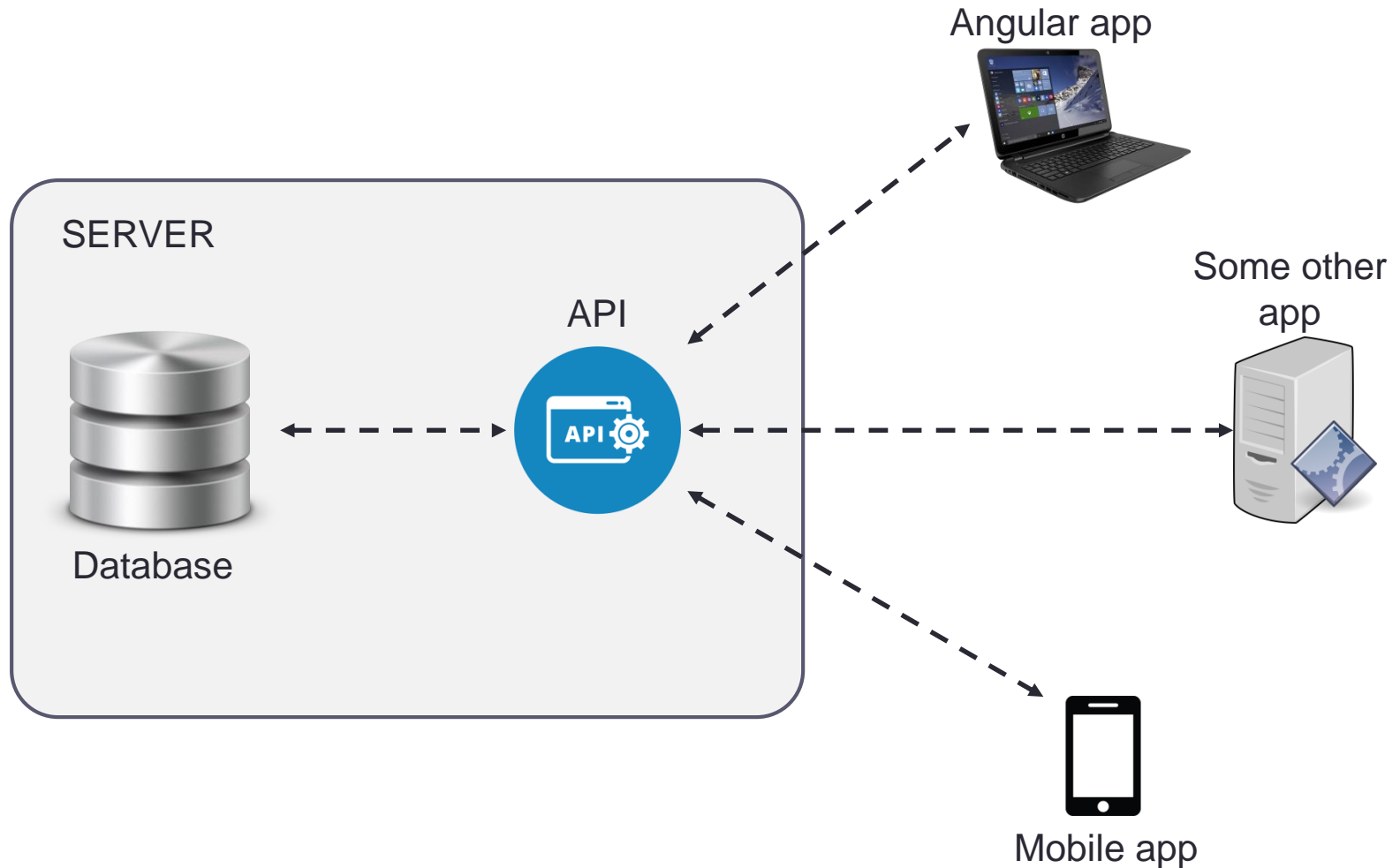
Request & Response



Where does Angular fit?



Where does Angular fit?



Angular CLI

- Toolset that makes creating, managing and building Angular apps very simple
- Great tool for big Angular projects
 - Website: <https://cli.angular.io>
 - Wiki: <https://github.com/angular/angular-cli/wiki>
- Requires Node.js
 - <https://nodejs.org>

```
> npm install -g @angular/cli  
> ng new my-first-app  
> cd my-first-app  
> ng serve
```

TypeScript

- Superset of JavaScript
 - Any valid JavaScript code is also valid TypeScript code
- Offers more features over vanilla JavaScript
 - Types, Classes, Interfaces, Modules, etc.
- TypeScript does not run in the browser, it is compiled to JavaScript (by CLI)
- Chosen as main language by Angular
- By far most documentation & example-base uses TypeScript
- Why TypeScript?
 - Strong Typing
 - reduces compile-time errors, provides IDE support
 - Next Gen JS Features
 - Modules, Classes, Import, Export, ...
 - Missing JS Features
 - Interfaces, Generics, ...

TypeScript

- Strong typing
 - More predictable and easier to debug
- Object-oriented features
 - Classes, Interfaces, Modules, Properties, Generics, etc.
- Compile-time errors
- Great tooling
 - Intelli-sense
- Installing TypeScript
 - `npm install -g typescript`
- Transpile TS to JS
 - `tsc some-program.ts`

TypeScript

- Declaring variables
 - var
 - let
- Types
 - number
 - string
 - boolean
 - any
- Type Assertions
 - `<string>message`
 - `message as string`
- Arrow Functions
 - `let log = (message) => console.log(message);`

TypeScript

- Interfaces

```
interface Point {  
    x: number,  
    y: number  
}
```

- Classes

```
class Point {  
    x: number;  
    y: number;  
  
    draw() {  
        console.log('x = ' + this.x + ', y = ' + this.y);  
    }  
}
```

TypeScript

- Constructors
 - Optional parameters
- Access Modifiers
 - public (default)
 - private
 - protected
- Access Modifiers in Constructor Parameters
- Properties
 - Getter
 - `get X() { return this.x; }`
 - Setter
 - `set X(value) { this.x = value; }`
- Modules
 - export
 - import

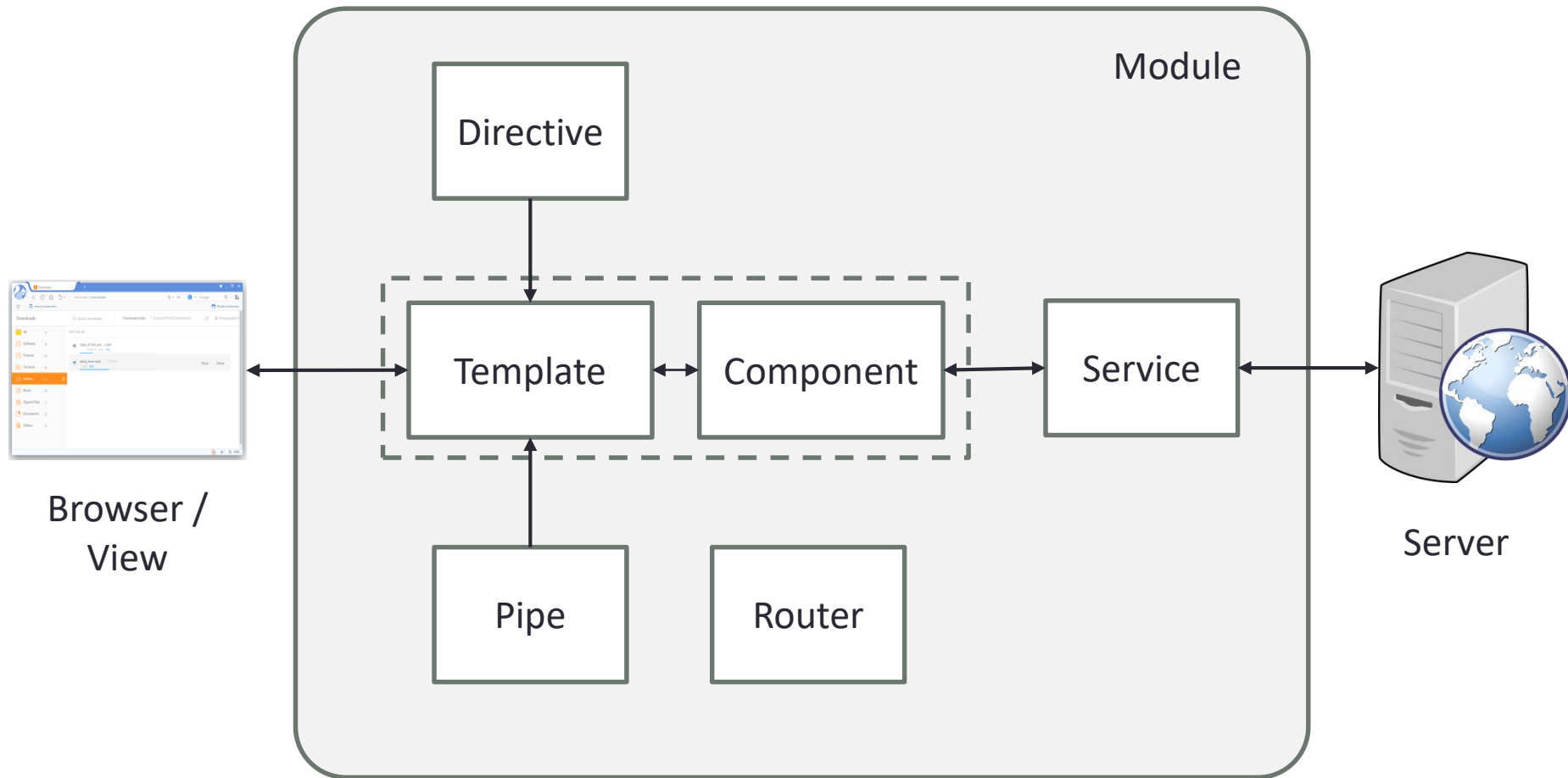
Bootstrap

- Add Bootstrap to the project
 - `npm install --save bootstrap`
- Add reference to bootstrap.css
 - `.angular-cli.json`
 - In “styles” array, add a reference to “bootstrap.min.css”
 - For e.g., `"../node_modules/bootstrap/dist/css/bootstrap.css"`
- How does an Angular app gets started?
 - `index.html` – Served by the server
 - `main.ts` – First file that gets executed
 - `app.module.ts` – Main loads this module
 - `app.component.ts`
 - Root component of the app
 - App module loads this component at the startup

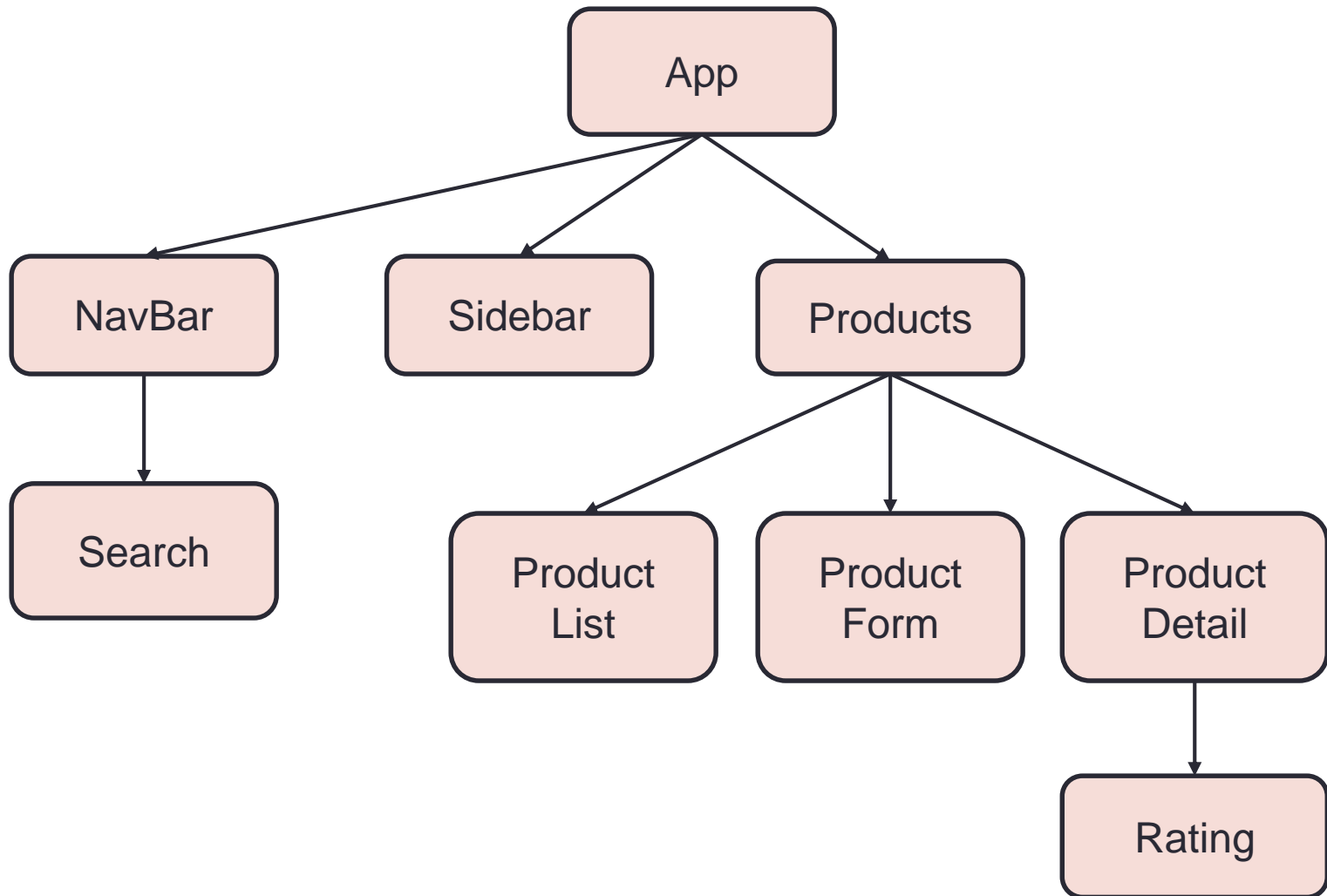
Angular Building Blocks

- Components
 - Encapsulates the template, data and the behavior of a view
 - Completely decoupled from DOM
- Directives
 - To modify DOM elements and/or extend their behavior
 - Built-in or custom
- Pipes
 - Takes in data as input and transforms it to a desired output
- Services
 - Encapsulates any non UI logic
 - Http calls, logging, business logic, etc
 - Any logic not related to a view is delegated to a service
- Routers
 - Responsible for navigation from one view to another
- Modules
 - A block of highly related classes

Angular Building Blocks



Components



Components

- Key feature of Angular
- Encapsulate the template, data and the behavior of a view
- Allows you to break a complex web page into smaller, manageable & reusable parts
- Plain TypeScript class
- App component
 - Root component
 - Holds our entire application
 - Other components are added to App component
- A Component has its own
 - Template – HTML markup
 - Style – CSS styles
 - Business logic (data and behavior) – TypeScript code
- Promotes
 - Reusability
 - Maintainability
 - Testability

Decorators

- Extends the behavior of a class / function without explicitly modifying it
- Attaches metadata to classes

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

@Component({
  selector: 'app-server',
  templateUrl: 'server.component.html'
})
export class ServerComponent {
}
```

Modules

- Organizes an app into cohesive blocks of functionality
- A class decorated with @NgModule metadata
- Every Angular app has at least one module class, the **root** module

```
@NgModule({
  imports: [module1, module2, ...],
  declarations: [
    component(s), directive(s), pipe(s), ...
  ],
  providers: [service1, service2, ...],
  bootstrap: [AppComponent]
})
export class AppModule{ }
```

Exercise

- Creating a new component
 - Create a new file, for e.g., products.component.ts
 - Create a class – ProductsComponent
- Understanding Decorator
 - Add decorator - @Component()
 - import { Component } from '@angular/core';
 - Provide metadata within @Component decorator
 - selector, templateUrl
- Understanding AppModule
 - Register ProductsComponent within 'declarations' array
 - Import ProductsComponent into AppModule
- Using a component
 - Use the selector <app-products></app-products> within app component template

Exercise

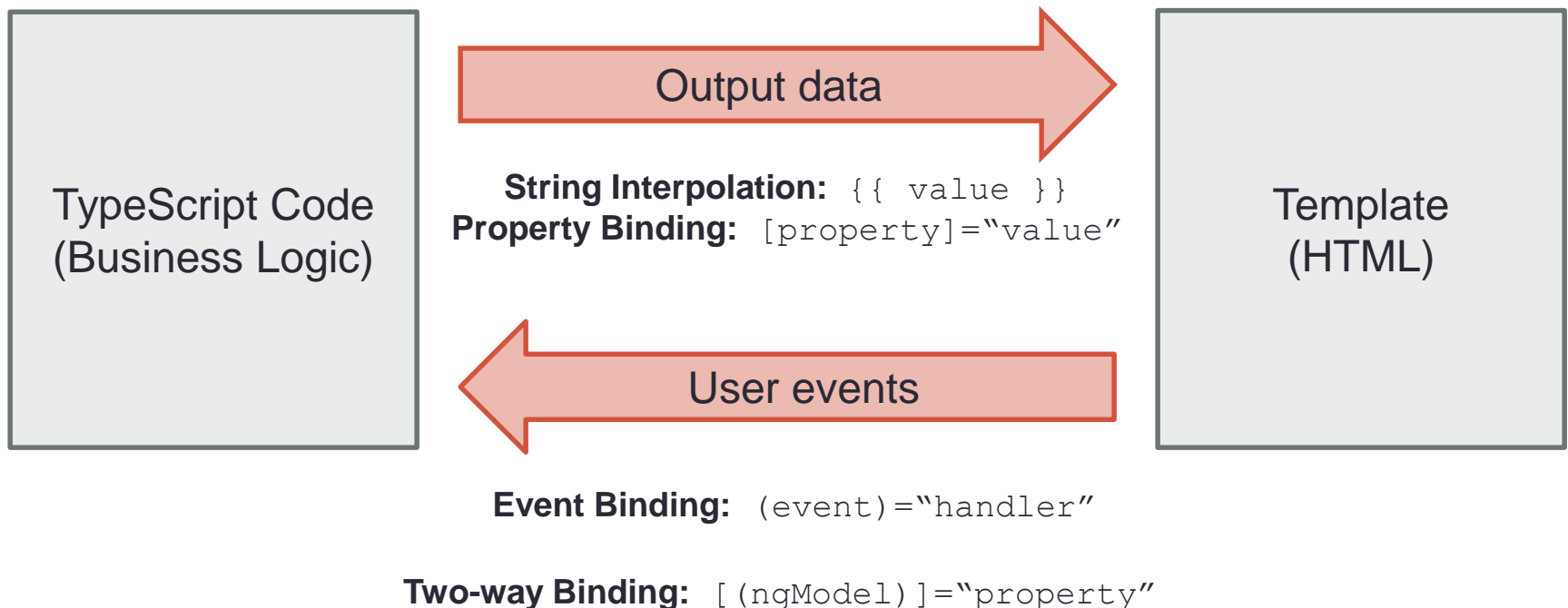
- Creating a component with CLI
 - ng generate component products
 - ng g c products

Component Templates & Styles

- Templates
 - templateUrl property – external template file
 - template property – inline template
- Styles
 - styleUrls property – external stylesheet file(s)
 - styles property – inline styles

Data Binding

- Communication between the TypeScript code and the HTML template



Data Binding

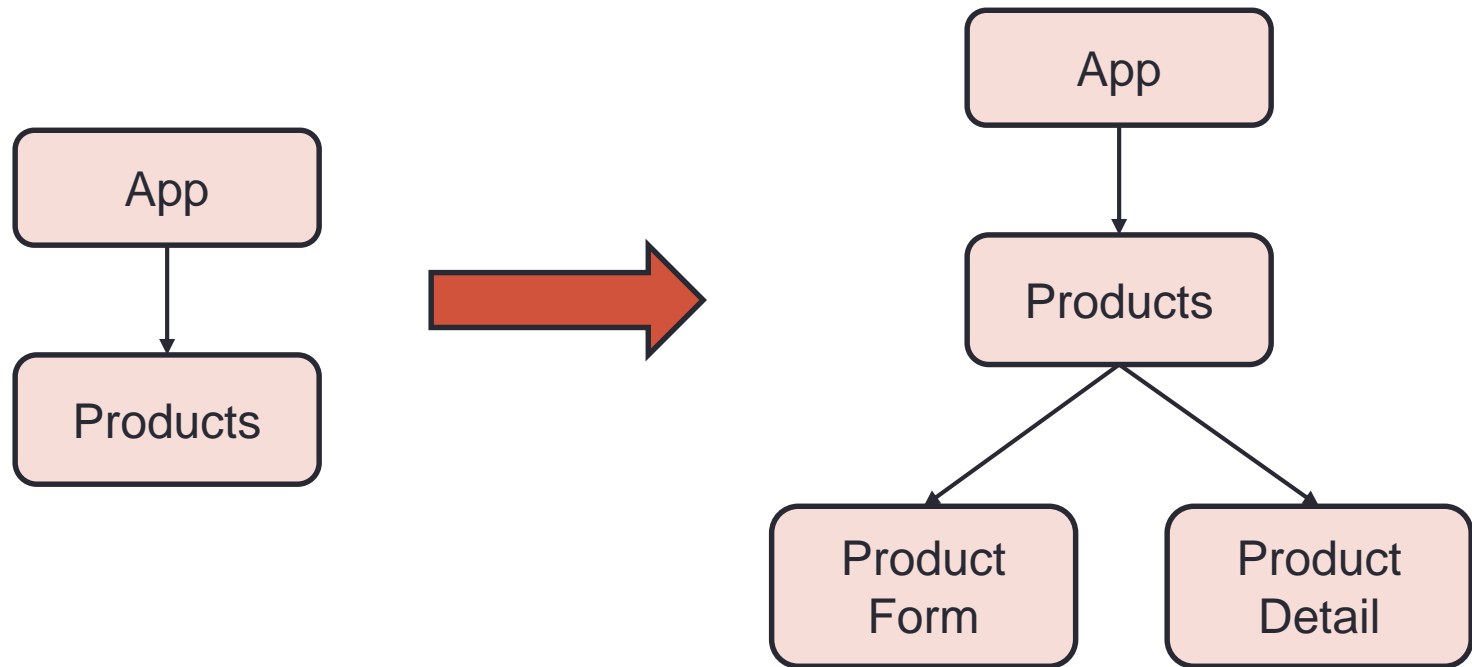
- String Interpolation
 - {{ }}
- Property Binding
 - []
- Event Binding
 - ()
 - \$event – Passing event data
- Two-way Data Binding
 - [(ngModel)]
 - Note: FormsModule should be imported in AppModule (imports[] array) to use ngModel

Directives

- Instructions in the DOM
- Components are directives with template
- Can be built-in or custom
- Built-in directives
 - Structural directives
 - Have a leading *
 - Alter layout by adding, removing, and replacing elements in DOM
 - E.g. *ngIf, *ngFor
 - Attribute directives
 - Look like a normal HTML attribute
 - Modifies the behavior of an existing element by setting its display value property and responding to change events
 - E.g. ngStyle, ngClass

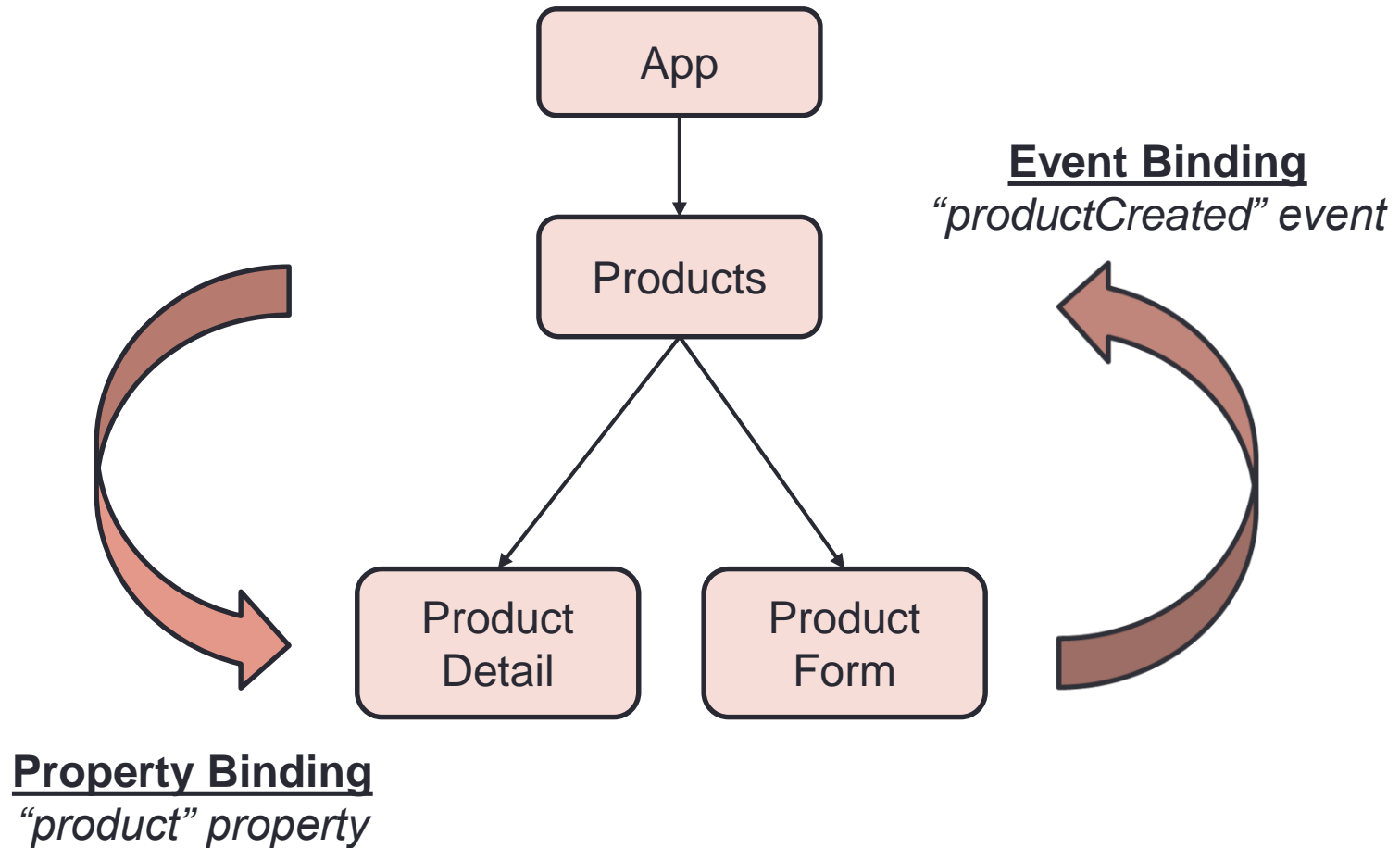
Component Interaction

- Splitting app into multiple components



Component Interaction

- Overview



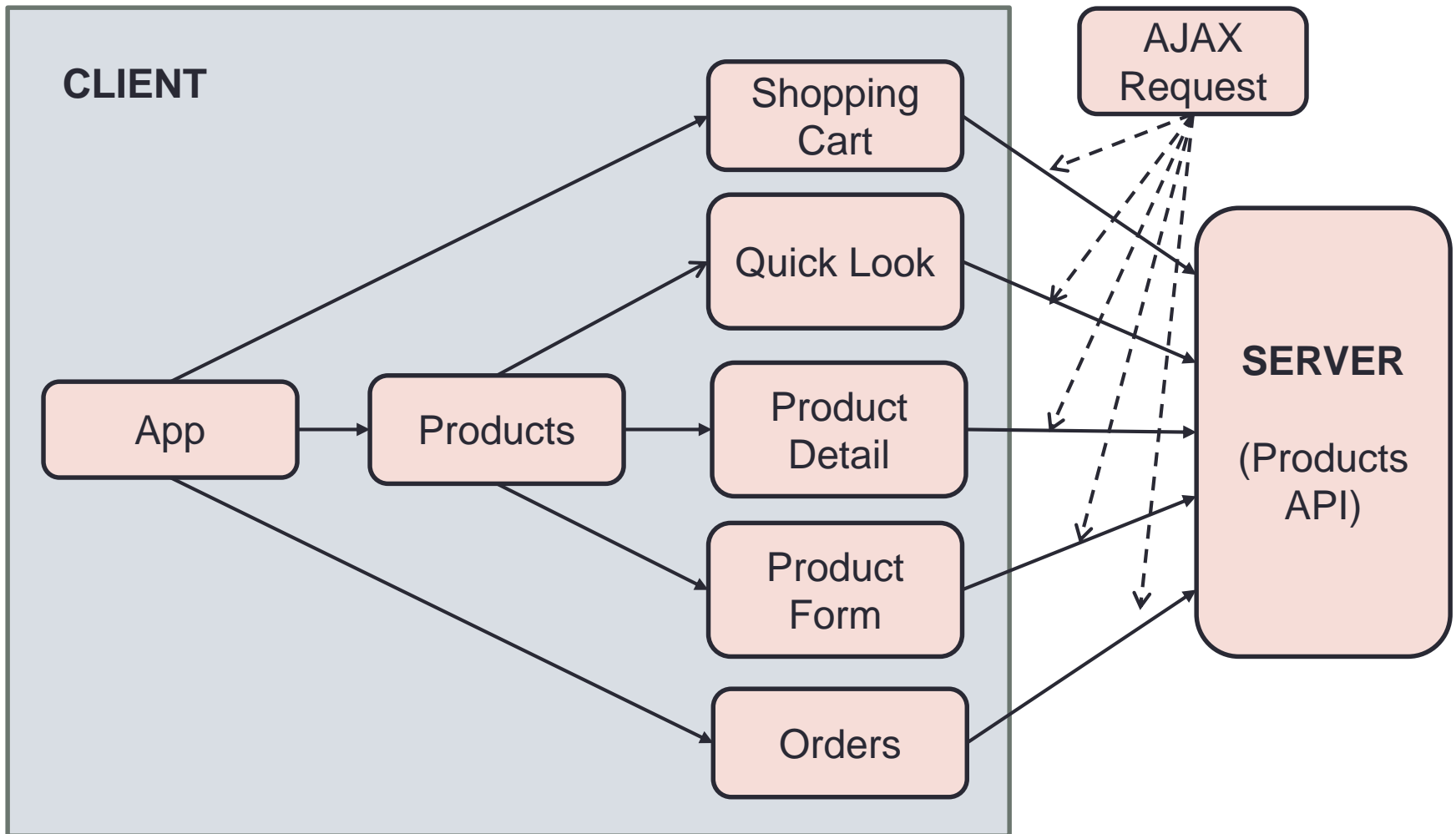
Component Interaction

- Binding to Custom Properties
 - Pass data from parent to child component
 - @Input() decorator
- Binding to Custom Events
 - Emitting event from child component
 - @Output() decorator
 - EventEmitter<T>
 - eventEmitterObj.emit()

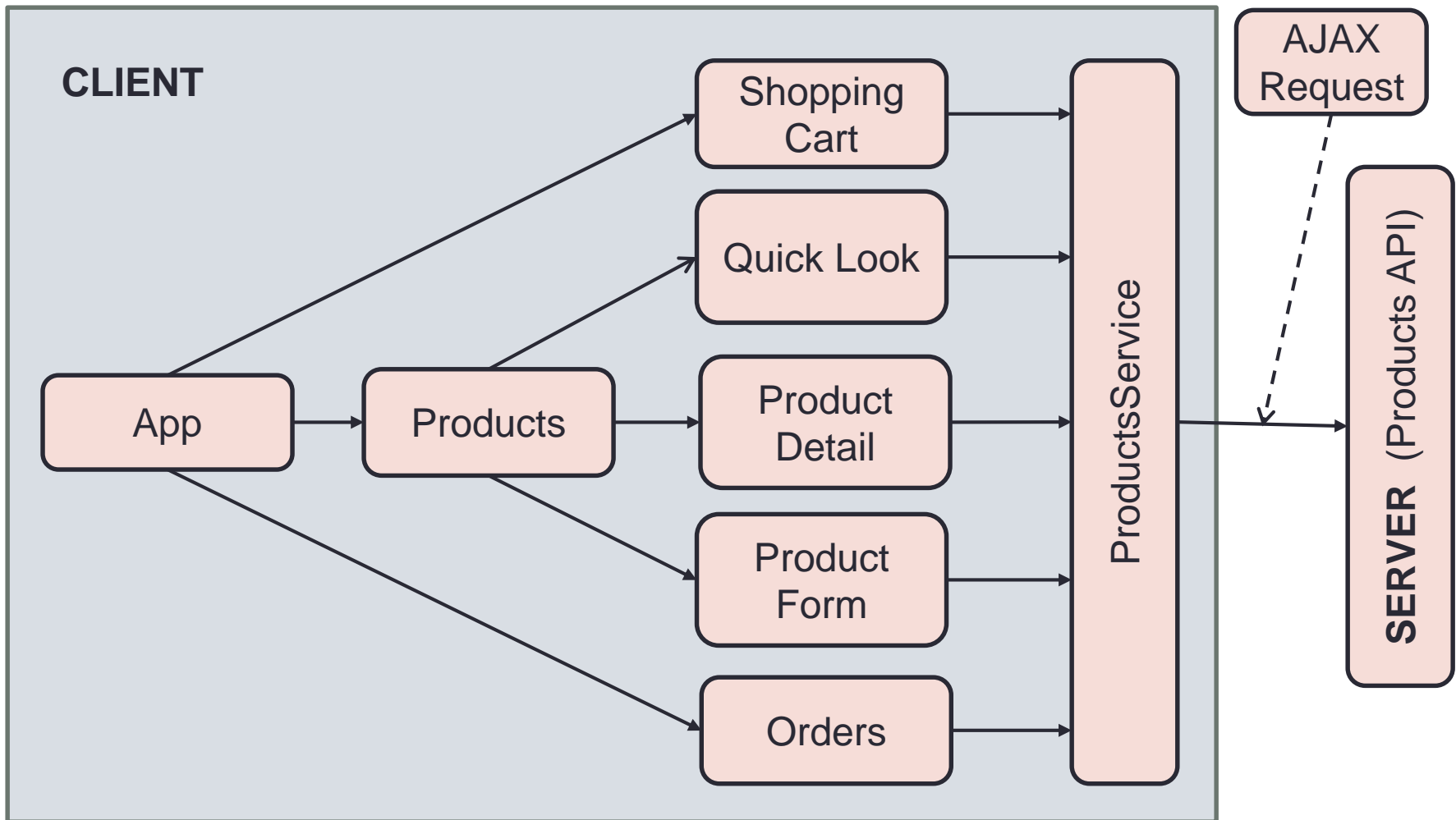
View Encapsulation

- Understanding View Encapsulation
- @Component()
 - encapsulation: ViewEncapsulation.None
- ViewEncapsulation
 - Emulated – default
 - Native
 - None

Services



Services



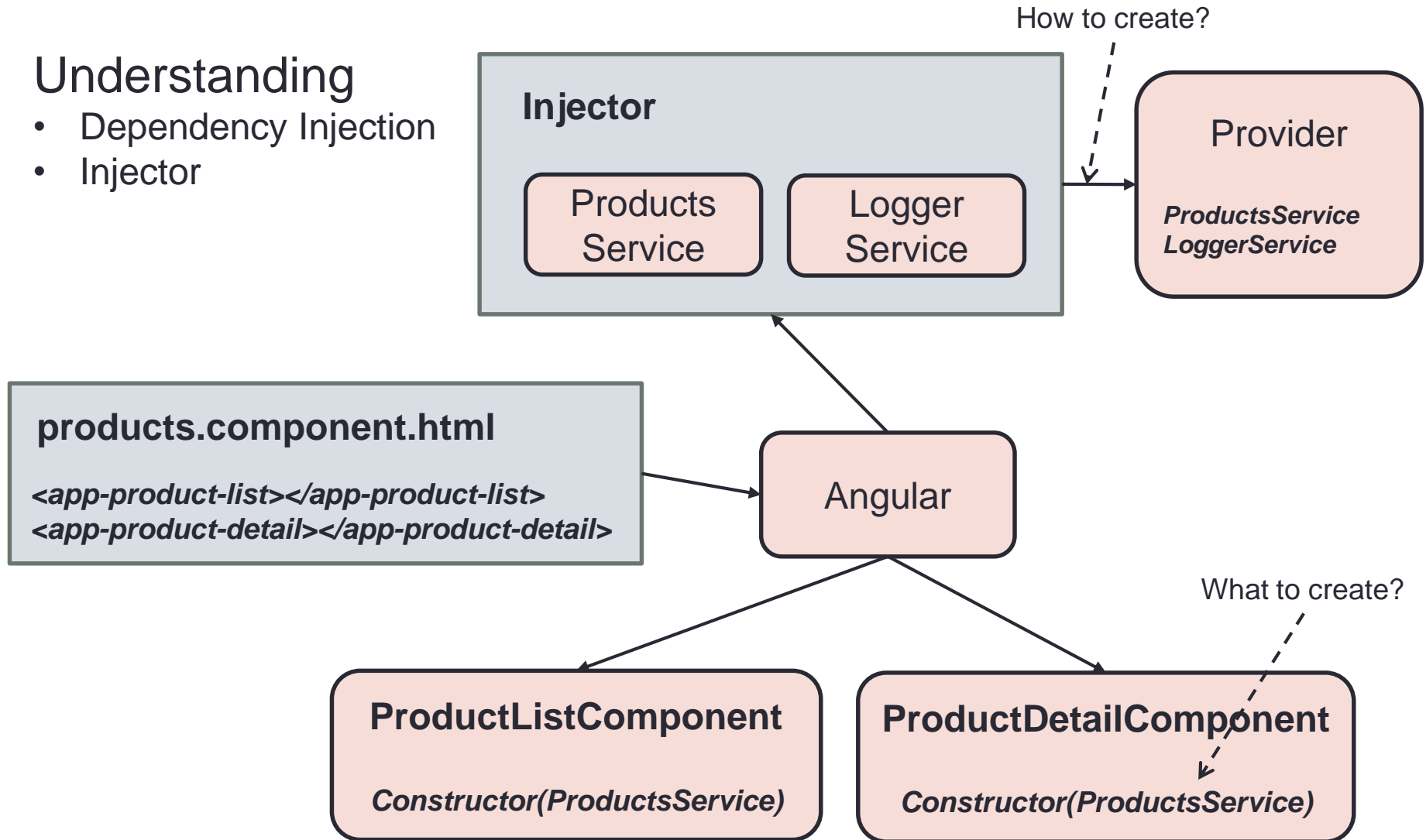
Services

- A class with a narrow, well-defined purpose
 - For e.g.
 - Logging service
 - Data service
 - Tax calculator
 - App configuration
 - Message bus
- Acts as a central repository/business unit
- Creating a service
- Injecting a service into a component
 - Constructor
 - Providers
 - Component level
 - Module level
- Injecting a service into another service
 - @Injectable()

Services

Understanding

- Dependency Injection
- Injector



Services

- Controlling the creation of instances of a Service

AppModule

Same instance of Service is available ***Application wide***

AppComponent

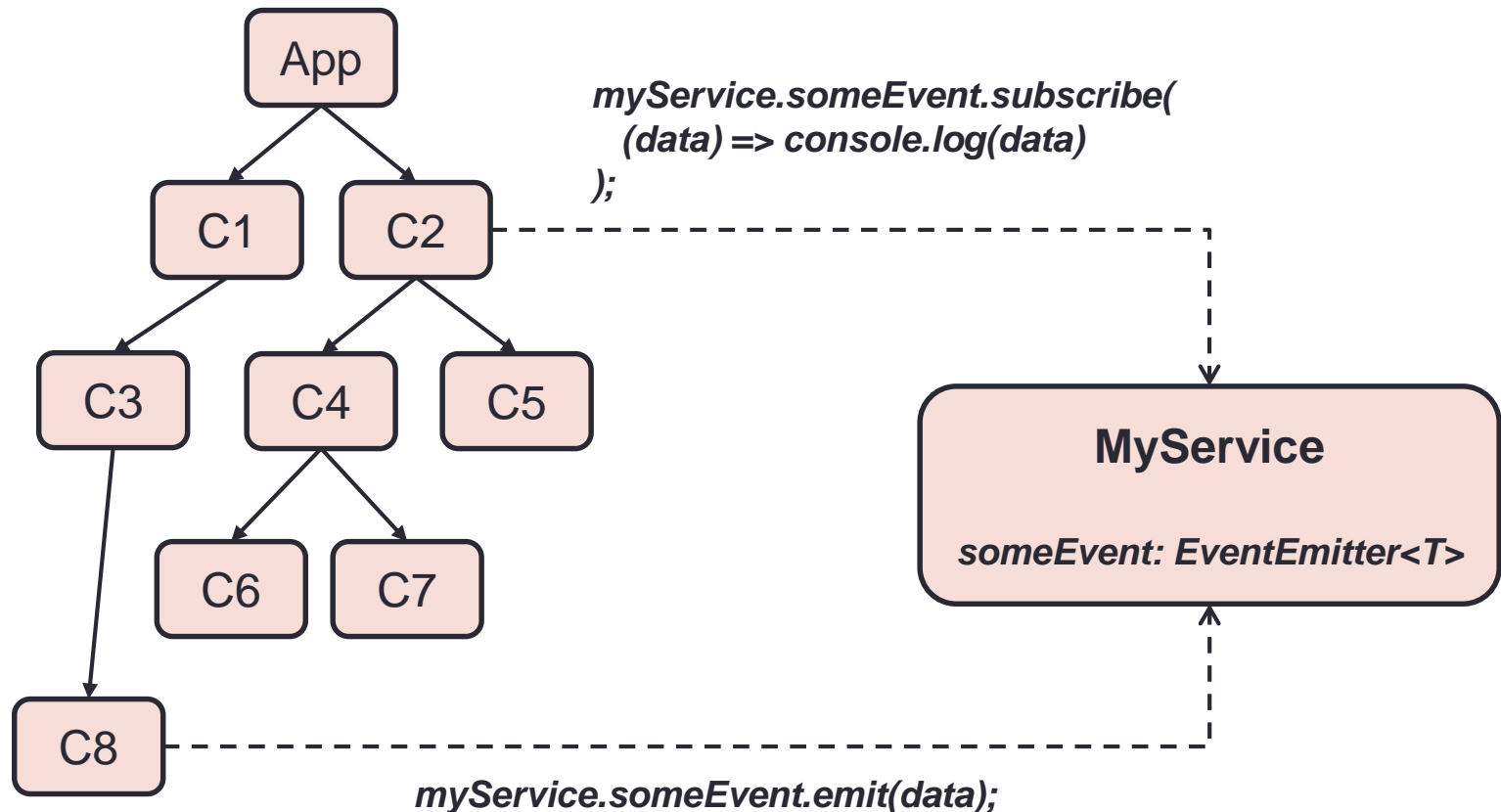
Same instance of Service is available for ***all Components*** (but not for other services)

**Any other
Component**

Same instance of Service is available for ***the Component*** and ***all its child Components***

Services

- Cross component communication using a service
 - In the service, expose an event object of type **EventEmitter**
 - From the source component, invoke **emit()** method, pass necessary data as an argument
 - From the destination component subscribe to the service's event object using **subscribe()** method, pass callback function as an argument



Routing

- Setting up routes (@angular/router module)
 - Routes
 - Define a constant appRoutes of type Routes
 - RouterModule.forRoot()
 - Register the routes with RouterModule.forRoot()
 - Include this in imports array of app module
- Loading Routes
 - <router-outlet> directive
- Navigating with Router Links
 - routerLink directive
- Styling active links
 - routerLinkActive="active"
 - [routerLinkActiveOptions]="{exact: true}"
- Navigating Programmatically
 - Import Router from @angular/router
 - Inject Router within the constructor
 - Router.navigate(['/products'])

Routing

- Passing Parameters to Routes
 - `[routerLink] = “['/servers', 10]”`
- Fetching Route Parameters
 - `ActivatedRoute.snapshot.params['id']`
 - `ActivatedRoute.params.subscribe()`
- Passing Query Parameters
 - `[queryParams] = “{allowEdit: true}”`
- Retrieving Query Parameters
 - `ActivatedRoute.snapshot.queryParams[]`
 - `ActivatedRoute.queryParams.subscribe()`
- Setting up Child Routes
- Redirecting and Wildcard Routes
- Outsourcing the Route Configuration

Observables

- Used to handle asynchronous tasks / operations
- Can be thought of as a data source
 - E.g. User input event, Http requests
- Object we import from a third-party package – rxjs
- Follows Observable pattern
 - Observable
 - Stream – timeline
 - Multiple events/data packages emitted by the observable, depending on the data source
 - Observer – your code
 - 3 ways of handling data packages
 - Handle Data
 - Handle Error
 - Handle Completion

Observables

- Observable – rxjs/Observable
 - Observable.interval()
 - Observable.create()
 - Observer - rxjs/Observer
 - Observer.next()
 - Observer.error()
 - Observer.complete()
 - Observable.subscribe() – returns Subscription (rxjs/Subscription)
- Subject
 - Subject.next()
 - Subject.subscribe()
- <http://reactivex.io/rxjs/>

Forms

- Angular helps
 - To get form values entered by the user
 - To check if the form is valid
 - To conditionally change the way the form is displayed
- Two Approaches
 - Template-Driven Forms
 - Angular infers the form object from the DOM (Template)
 - Good for simple forms
 - Simple validation
 - Easier to create
 - Less code
 - Reactive Forms
 - Form is created programmatically and synchronized with the DOM
 - Good for complex forms
 - More control over validation logic
 - Unit testable

Forms

- Template-Driven Forms
 - Make sure that FormsModule is imported within the app module
 - Creating the form
 - `<form>` tag need not have these attributes:
 - action
 - method
 - Registering the controls
 - Include “ngModel” directive
 - Include “name” attribute
 - Submitting the form
 - Include ngSubmit event
 - `<form (ngSubmit)="onSubmit(f)" #f="ngForm">`
 - Understanding form state
 - Accessing the form with @ViewChild
 - `@ViewChild('f') productForm: NgForm;`
 - User Input Validation
 - Directives
 - required, email, minlength, maxlength, pattern
 - Form State
 - pristine / dirty, touched / untouched, valid / invalid
 - CSS
 - ng-pristine / ng-dirty, ng-touched / ng-untouched, ng-valid / ng-invalid

Forms

- Template-Driven Forms

- Using Form State

- Disable submit button

- ```
<button type="submit" [disabled]="!f.valid">Save</button>
```

- Include CSS classes to provide better user feedback & experience

- ```
input.ng-invalid.ng-touched {  
  border: 1px solid red;  
}
```

- Display validation messages

- ```

 Product name is required.

```

- Using ngModel with one-way and two-way binding

- [ngModel]="productName"
      - [(ngModel)]="productName"

- Grouping Form Controls

- ngModelGroup="address"
      - #addr="ngModelGroup"

- Using Form Data

- productForm.value

- Resetting Forms

- productForm.reset()

# Forms

- Reactive Forms
  - More control over form structure and behavior
  - Easier to unit test
  - The form is created programmatically
    - `signupForm: FormGroup`
    - `FormGroup` is imported from `'@angular/forms'`
  - AppModule
    - Import `ReactiveFormsModule` from `'@angular/forms'`
    - Add `ReactiveFormsModule` to `'imports'` array within `@NgModule` decorator
  - Creating a form in code
    - Preferably use `'ngOnInit()'` to create the form

```
this.signupForm = new FormGroup({
 'username': new FormControl('default-value', validator),
 'email': new FormControl('default-value', validator)
});
```

# Forms

- Reactive Forms
  - Linking HTML and Form
    - Use 'formGroup' directive to link `<form>` and form object
      - `<form [formGroup]="signupForm">`
    - Use 'formControlName' directive to link form control and form object's property
      - `<input type="text" id="username" formControlName="username">`
  - Submitting the form
    - Use 'ngSubmit' event
      - `<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">`
  - Adding Validation
    - Pass validator(s) as second parameter to FormControl object

```
this.signupForm = new FormGroup({
 'username': new FormControl(null, Validators.required),
 'email': new FormControl(null, [Validators.required, Validators.email])
});
```
    - Import Validators from '@angular/forms'

# Forms

- Reactive Forms

- Getting access to controls

- formObj.get('control-name')

- E.g.

- ```
<span *ngIf="!signupForm.get('email').valid" class="help-block">
```

- Email is required.

- ```

```

- Showing Specific Validation Errors

- ```
<div class="help-block"
```

- ```
 *ngIf="email.touched && email.invalid">
```

- ```
    <div *ngIf="email.errors.required">
```

- Email is required.

- ```
 </div>
```

- ```
    <div *ngIf="email.errors.email">
```

- Email is not valid.

- ```
 </div>
```

- ```
</div>
```

Forms

- Reactive Forms

- Implementing Custom Validators

- validate(c: AbstractControl): ValidationErrors|null***

- AbstractControl, ValidationErrors – '@angular/forms'

- Asynchronous Validators

- validate(c: AbstractControl):***

- Promise<ValidationErrors|null>|Observable<ValidationErrors|null>***

- Showing a loader image

- `<div *ngIf="name.pending">`

- `Checking for uniqueness...`

- `</div>`

Forms

- Reactive Forms
 - Validating the Form Upon Submit
 - Component

```
this.form.setErrors({
  invalidCustomer: true
});
```
 - Template

```
<div class="alert alert-danger" *ngIf="form.errors">
  Customer data is invalid!
</div>
```

Forms

- Reactive Forms

- Nested FormGroup

```
this.form = new FormGroup({  
  name: new FormControl("", Validators.required),  
  contact: new FormGroup({  
    phone: new FormControl("", Validators.required),  
    email: new FormControl("", Validators.required)  
  }),  
  city: new FormControl("", Validators.required)  
});
```

- Template

```
<div formGroupName="contact">  
  <div class="form-group">  
    <label for="customerPhone">Phone</label>  
    <input type="text" class="form-control"  
      id="customerPhone" name="customerPhone" formControlName="phone">  
  </div>  
  <div class="form-group">  
    <label for="customerEmail">Email</label>  
    <input type="text" class="form-control"  
      id="customerEmail" name="customerEmail" formControlName="email">  
  </div>  
</div>
```

Forms

- Reactive Forms
 - Create Forms using FormBuilder

```
export class CustomerFormComponent {  
  form: FormGroup;  
  
  constructor(private fb: FormBuilder) {  
    this.form = this.fb.group({  
      name: ['', Validators.required],  
      contact: this.fb.group({  
        phone: ['', Validators.required],  
        email: ['', Validators.required]  
      }),  
      city: ['', Validators.required]  
    });  
  }  
}
```


Pipes

- Transform output, do not modify the underlying data
- Format the value of an expression for display
- Built-in pipes
 - uppercase
 - date
- Using pipes
- Parameterizing pipes
- Chaining multiple pipes

Pipes

- Creating custom pipes
 - Create a class
 - Implement PipeTransform interface (`@angular/core`)
 - Implement `transform()` method
- Parameterizing a custom pipe

Server Communication

- Http – '@angular/http'
 - Performs http requests using XMLHttpRequest
- Getting Data
 - App Module
 - Import 'HttpModule' from '@angular/http'
 - Add 'HttpModule' to 'imports' array
 - Constructor
 - Inject 'Http' instance in the constructor
 - Import 'Http' from '@angular/http'
 - Get data
 - Use Http.get('url') method to create the get request
 - Http.get() returns Observable<Response>
 - Use subscribe() method of the Observable
 - responseObj.json() with actually return the data
- Creating Data
 - Http.post('url', newObject)
 - The Response object contains the newly created object

Server Communication

- Updating Data
 - `Http.put('url' + id, updatedObject)`
 - `Http.patch('url' + id, updatedObject)`
 - The Response object contains the updated object
- Deleting Data
 - `Http.delete('url' + id)`
- OnInit Interface
 - Constructor should be lightweight and should not perform expensive operations
 - Do not call Http services in the constructor of the component
 - Use `OnInit.ngOnInit()` method for initialization

Server Communication

- Separation of Concerns
 - Single responsibility
 - Do not include http service calls in the component
- Handling Errors
 - Unexpected errors
 - Server is offline
 - Network is down
 - Unhandled exceptions
 - Expected errors
 - Not found error (HTTP error code 404)
 - Bad request error (HTTP error code 400)

Server Communication

- The Catch operator

```
import 'rxjs/add/operator/catch';
```

```
return this.http.get('api-url')  
  .catch( (error: Response) => console.log(error.message) );
```

- Throw application errors

```
import { Observable } from 'rxjs/Observable';  
import 'rxjs/add/observable/throw';
```

```
return this.http.get('api-url')  
  .catch( (error: Response) => {  
    return Observable.throw(new AppError(error));  
  })
```

Server Communication

- Global Error Handling

- Create a class – `AppErrorHandler` – that implements `ErrorHandler` from `'@angular/core'`
- Implement `'handleError()'` method in this class
- In the app module, register `'AppErrorHandler'` in `'providers'` array

```
providers: [  
  { provide: ErrorHandler, useClass: AppErrorHandler }  
]
```

- The Map Operator

```
import 'rxjs/add/operator/map';
```

```
return this.http.get('api-url')  
  .map( response => response.json() );
```

Server Communication

- Observables vs Promises

- Observables

- Lazy
 - Can be converted into promises using toPromise() operator
 - Handle multiple values over time
 - Cancellable

- Promises

- Eager
 - Do not have operators like in Observables
 - Called only once and will return a single value
 - Not cancellable

Q & A

- Thank you!