ngular

Dr. Ernesto Lee, Miami Dade College
Professor Sandrilla Washington, Spelman College

npm QuickStart

Node.js package manager

Installing Dependencies

Package manager

Global Installs

```
$ npm install typescript --global //-g for short
```

Global Packages Location

$ npm get prefix

Mac:
/Users/[username]/.npm-packages/lib/node_modules

PC:
%AppData%\npm\node_modules (Windows 7)
%USERPROFILE%\Application Data\npm\node_modules (Windows XP)

Local Installs

```
$ npm init –y #creates package.json –y uses defaults

$ npm install typescript --save-dev #saves in package.json
                                    #creates node_modules directory

$ tsc -v #fails because can't find package

$ node_modules/.bin/tsc -v #outputs version info
```

package.json
dependencies and devDependencies

$ npm install rxjs --save

```json
{
    "name": "npm-quickstart",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {},
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "rxjs": "^5.2.0"
    }
}
```

$ npm install typescript --save-dev

```json
{
    "name": "npm-quickstart",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {},
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "rxjs": "5.2.0"
    },
    "devDependencies": {
        "typescript": "^2.2.1"
    }
}
```

Semantic Versioning

- Major.Minor.Patch

- If you were starting with a package 1.0.4, this is how you would specify the ranges:

  - Patch releases: ~1.0.**4**

  - Minor releases: **^**1.**0**.4

  - Major releases: *** or x | 1**.0.4 → 2.0.0

Sharing Dependencies

```
$ npm install //installs dependencies and devDependencies
$ npm install --production //installs dependencies only
```

Updating Dependencies

```
$ npm update lite-server -g //update a global dependency
$ npm update rxjs --save //update a local dependency
```

Uninstalling Dependencies

```
$ npm uninstall lite-server -g //uninstall a global dependency
$ npm uninstall rxjs --save //uninstall a local dependency
```

Lab

Node Package Manager (npm)

Open **TypeScriptLabManual.pdf** and follow the directions to do the following sections:

Create Project

Install TypeScript & RxJS

Understanding package-lock.json

- In an ideal world, the same **package.json** should produce the exact same **node_modules** tree, at any time

- In some cases, this is indeed true. But in many others, npm is unable to do this

- To **reliably** produce the **exact node_modules tree**, **package-lock.json** was created.

Broken package.json scenarios

- Different versions of npm (or other package managers) may have been used to install a package, each using slightly different installation algorithms.
- A new version of a direct semver-range package may have been published since the last time your packages were installed, and thus a newer version will be used.
- **A dependency of one of your dependencies may have published a new version, which will update even if you used pinned dependency specifiers (1.2.3 instead of ^1.2.3)**
- The registry you installed from is no longer available, or allows mutation of versions (unlike the primary npm registry), and a different version of a package exists under the same version number now.

Using npm as a Build Tool

Build Automation | npm scripts

What about Grunt and Gulp?

- npm's scripts directive can do everything that these build tools can, more succinctly, more elegantly, with less package dependencies and less maintenance overhead

- What's Wrong with Gulp and Grunt?

   1. Complexity
   2. Dependence on plugin authors
   3. Frustrating debugging
   4. Disjointed documentation

Your First Script

```
$ npm init -y //creates package.json
{
  "name": "npmscriptsdemo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "hi": "echo hello world "          Custom npm script
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

$ npm run-script hi //hello world      Run the custom script
```

Shortcuts

Instead of:

```
$ npm run-script [script-name]
```

You could use the shorter:

```
$ npm run [script-name]
```

Or if it's one of the npm supported scripts you can omit the run command:
```
$ npm [script-name]
```

Supported Scripts

- prepublish: Run BEFORE the package is published. (Also run on local **npm install** without any arguments.)
- publish, postpublish: Run AFTER the package is published.
- preinstall: Run BEFORE the package is installed
- install, postinstall: Run AFTER the package is installed.
- preuninstall, uninstall: Run BEFORE the package is uninstalled.
- postuninstall: Run AFTER the package is uninstalled.
- preversion, version: Run BEFORE bump the package version.
- postversion: Run AFTER bump the package version.
- pretest, test, posttest: Run by the **npm te**st command.
- prestop, stop, poststop: Run by the **npm stop** command.
- prestart, start, poststart: Run by the **npm start** command.
- prerestart, restart, postrestart: Run by the **npm restart** command.
  - Note: npm restart will run the stop and start scripts if no restart script is provided.

Running Local Node Modules

```
$ npm init #creates package.json

$ npm install typescript --save-dev #saves in package.json
                              #creates node_modules directory


$ tsc -v #fails because can't find package

$ node_modules/.bin/tsc -v #returns version

//package.json snippet
"scripts": {
  "tsc": "tsc"
}

$ npm run tsc -- -v #returns version but path not required in package.json
```
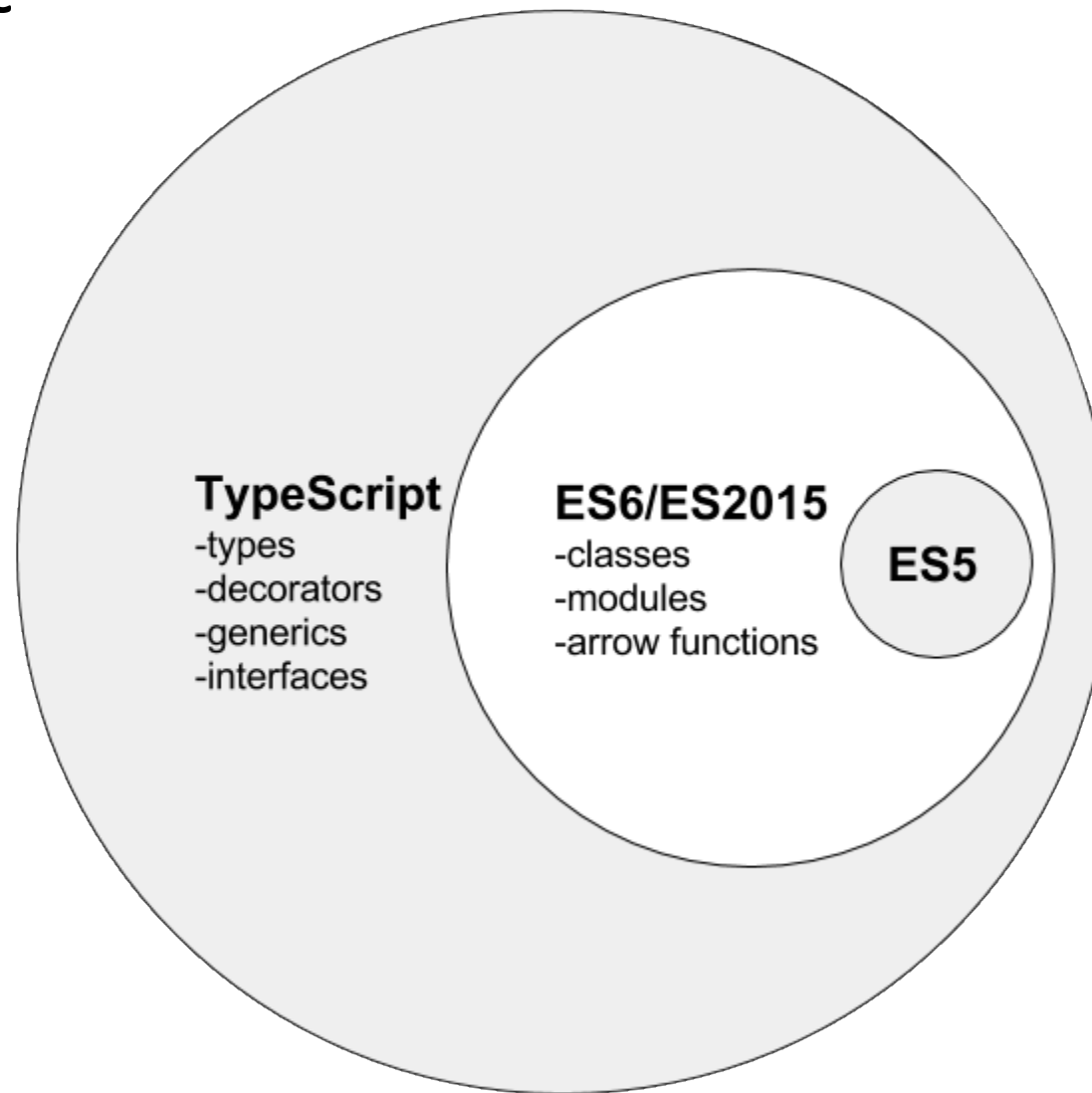
TypeScript/ES2015

Introduction

TypeScript

How TypeScript Works

Why TypeScript?

- TypeScript is a primary language for Angular application development
  - Angular is written in TypeScript.
- Recommended by Angular core team and Google
- Types enable better tooling including
  - Refactoring
  - Navigating code
  - Code completion
- Decorators (annotations) provide an easily understandable API

Who is Behind TypeScript?

Anders Hejlsberg

Core developer of TypeScript

Microsoft technical fellow

Lead architect C#

Original author Turbo Pascal: Delphi

Image from Wikimedia Commons

Installing TypeScript

Global

```
npm install -g typescript
```

Local

```
npm init //creates package.json
npm install typescript --save-dev //saves version to package.json
//To ensure you're getting local version run using npm scripts
```

Configuring TypeScript

```
tsc --init
message TS6071: Successfully created a tsconfig.json file.


//default tsconfig.json
{
    "compilerOptions": {
        "target": "es5",

        ...
    }
}
```

Compiling with TypeScript

```
tsc
//OR
tsc --watch
```

JavaScript is valid TypeScript

Q: What would be the output of the following JavaScript code run through the TypeScript compiler?

```
function greeter(name) {
    return "Hello " + name;
}

console.log(greeter("Anders"));
```

A: The same code.

Type Annotations

```typescript
function greeter(name: string) {
    return "Hello " + name;
}

console.log(greeter("Anders"));
```

Lab

Node Package Manager (npm) & TypeScript

Open **TypeScriptLabManual.pdf** and follow the directions to do the following sections:

Run TypeScript

Type Annotations

ES2015 Classes

```
class Student{
    public firstName: string;        ← Property: public is the default
    middleInitial: string;
    lastName: string;                ← Public keyword is not required

    constructor(firstName: string,   ← Constructor
                middleInitial: string,
                lastName: string){

        this.firstName = firstName;
        this.middleInitial = middleInitial;
        this.lastName = lastName;
    }

    getFullName(){
        return this.firstName + " " + this.middleInitial + ". " + this.lastName;
    }
}
let student = new Student("John", "D", "Rockefeller");   ← Create instance and invoke method
console.log(student.getFullName());
```

TypeScript Automatic Property Assignment

```typescript
class Student {
    firstName: string;
    middleInitial: string;
    lastName: string;

    constructor(firstName, middleInitial, lastName) {
        this.firstName = firstName;
        this.middleInitial = middleInitial;
        this.lastName = lastName;
    }
}
--------------------------------------------------------------
class Student {

    constructor(public firstName:string,
                public middleInitial:string,
                public  lastName:string) {}

}
```

Equivalent

Classes & Interfaces

```typescript
interface Person {
    firstName: string;
    lastName: string;
}


class Student implements Person{
    fullName: string;
    constructor(public firstName, public middleInitial, public lastName) {
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}



function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let student = new Student("Jon", "M.", "Turner");
console.log(greeter(student));
```

Interface

Class that implements interface

Function that takes parameter of the interface type

Create instance of class and invoke method

Scope: var, let, const

- var
  - Function scope

- let
  - Block scope

- const
  - Block scope
  - Cannot change or be redeclared

ES2015 Arrow Functions

```
let evens = [0,2,4,6,8];

//verbose
// let odds = evens.forEach(function(v){
//      return v+1;
// });

//terse
let odds = evens.forEach(v => v + 1);

odds.forEach(line=> console.log(line)); //1,3,5,7,9
```

Lab

Node Package Manager (npm) & TypeScript

Open **TypeScriptLabManual.pdf** and follow the directions to do the following sections:

Classes

Scope
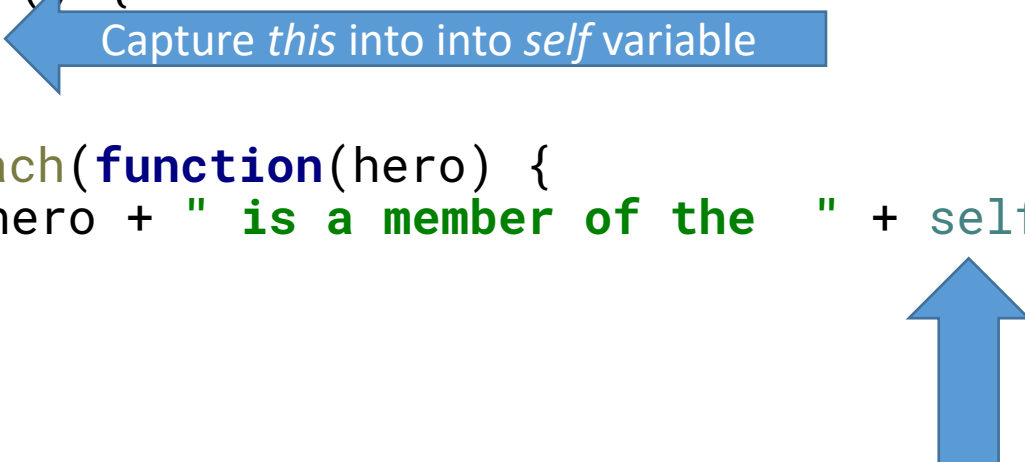
Arrow Functions

ES2015 Arrow Functions (this)

without an arrow function

```javascript
var organization={
    name: "Avengers",
    heroes: ["Hulk", "Iron Man", "Captain America"],
    printHeroes: function() {
        var self = this;          ⬅ Capture this into into self variable

        this.heroes.forEach(function(hero) {
            console.log(hero + " is a member of the  " + self.name + ".");
        });
    }
};

organization.printHeroes()
```

this.name is undefined so
we have to use self.name

ES2015 Arrow Functions (this)

with an arrow function

```javascript
var organization={
    name: "Avengers",
    heroes: ["Hulk", "Iron Man", "Captain America"],
    printHeroes: function() {
      this.heroes

        .forEach(h => console.log(h + " is a member of the " + this.name + "."));
    }
};



organization.printHeroes();
```

- *this.name is "Avengers"*
- *Arrow functions don't create a new function context*

TypeScript Decorators

- With the introduction of Classes in ES2015, there now exist certain scenarios that require additional features to support metadata modifying classes and class members. Aspect-Oriented Programming (AOP).

- Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members

- Decorators are a stage 1 proposal for JavaScript and are available as an experimental feature of TypeScript
  - They may change in future releases

TypeScript Decorators: Configuring

```json
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "experimentalDecorators": true
    }
}
```

Decorators

```typescript
function ClassLogger(target: Function) {
    console.log("The function that created this class is: " + target);
}

@ClassLogger
class Customer {
}

var customer = new Customer();
```
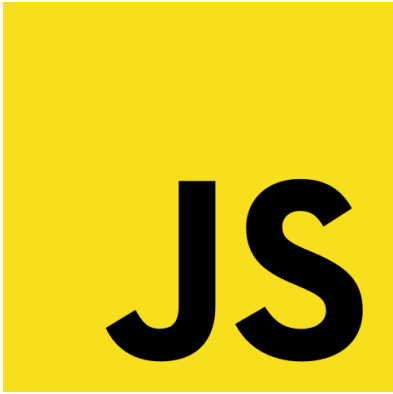
ES Modules

ES Modules

ES Modules Explained

- An ES6 module is a file containing JS code.
  - There's no special module keyword
  - A module reads like a script except:
    - ES modules are automatically strict-mode code
    - You can use import and export in modules
    - A JavaScript file is a module if it contains the import and/or export keyword
- Modules are executed within their own scope, not in the global scope
- Note: This feature is not implemented in any browsers natively at this time. It is implemented in by module loaders.

ES Modules Syntax

- The **export statement** is used to export functions, objects or primitives from a given file (or *module*)

- The **import statement** is used to import functions, objects or primitives that have been exported from an external module, another script, etc.

ES Module Example

```
//my-module.ts

export function myFunction(){
    return "myFunction was run.";
}

-------------------------------------------------------------------

//program.ts
import {myFunction} from "./my-module";
console.log(myFunction());
```

ES Module Privacy Example

```typescript
//my-module.ts

export function myFunction(){
    return "myFunction was run.";
}


function myPrivateFunction(){
    return "myPrivateFunction was run.";
}

-----------------------------------------------------------

//program.ts
import {myFunction, myPrivateFunction} from "./my-module";


//Module has no exported member myPrivateFunction.
```

ES Module Exporting

export function, object, primitive, and class

```ts
//my-module.ts

export function myFunction(){
    return "myFunction was run.";
}


var myObject = {
    name: 'I can access myObject\'s name',
    myMethod: function(){return 'myMethod on myObject is running.';}
}
export {myObject}


export const myPrimitive = 55;


export class MyClass{
    myClassMethod(){
        return "myClassMethod on myClass is running."
    }
}
```

ES Module Importing

import function, object, primitive, and class

```
//program.ts

import {myFunction, myObject, myPrimitive, MyClass} from "./my-module";


console.log(myFunction());


console.log(myObject.name);


console.log(myObject.myMethod());

console.log(myPrimitive);

let myClass = new MyClass();
console.log(myClass.myClassMethod());
```

ES2015 Template literals

```
class Student{

    constructor(private firstName: string,
                private middleInitial: string,
                private lastName: string){}


    getFullName(){
        return `
            First: ${this.firstName}
            Middle: ${this.middleInitial}
            Last: ${this.lastName}`;
    }

}
```

Use backticks `` | NOT single quotes ''

Lab

TypeScript

Open **TypeScriptLabManual.pdf** and follow the directions to do the following sections:

Modules

Do the remaining sections in the manual if time permits

Review (optional)

TypeScript

TypeScript Review

- Make a quick written list of 4-6 language features that TypeScript or ES6/ES2015 adds to JavaScript (ES5).

- Draw a set diagram like the one I showed earlier to show the relationship between ES5, ES6/ES2015, and TypeScript.

Angular Overview

Big Picture

Why Angular

Why would we build our application front-end in JavaScript?

Why would we use a single-page application framework?

What is your current understanding?

Why a single-page application?

- The user experience of a desktop or native application
- The deployment story of a web application
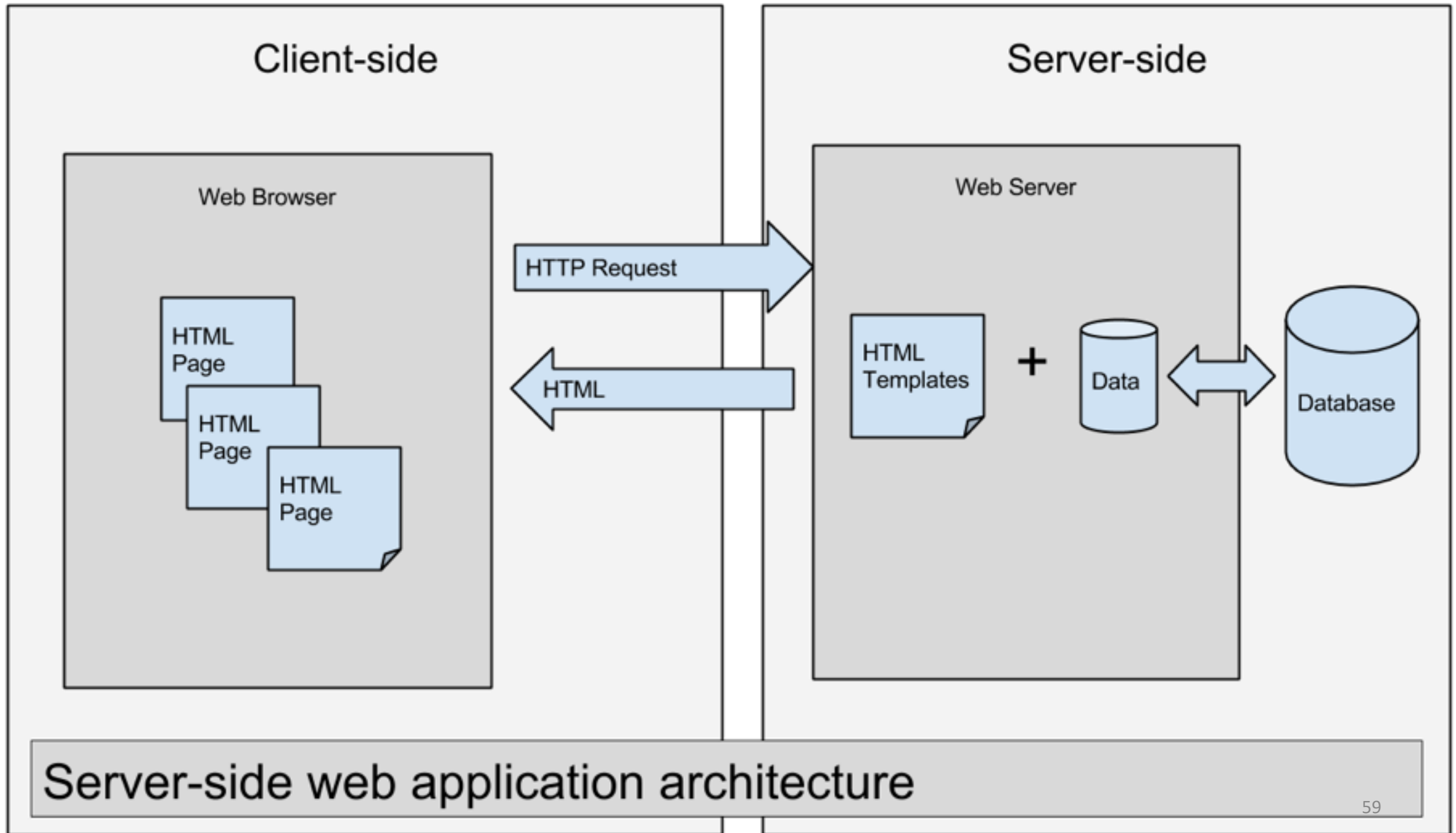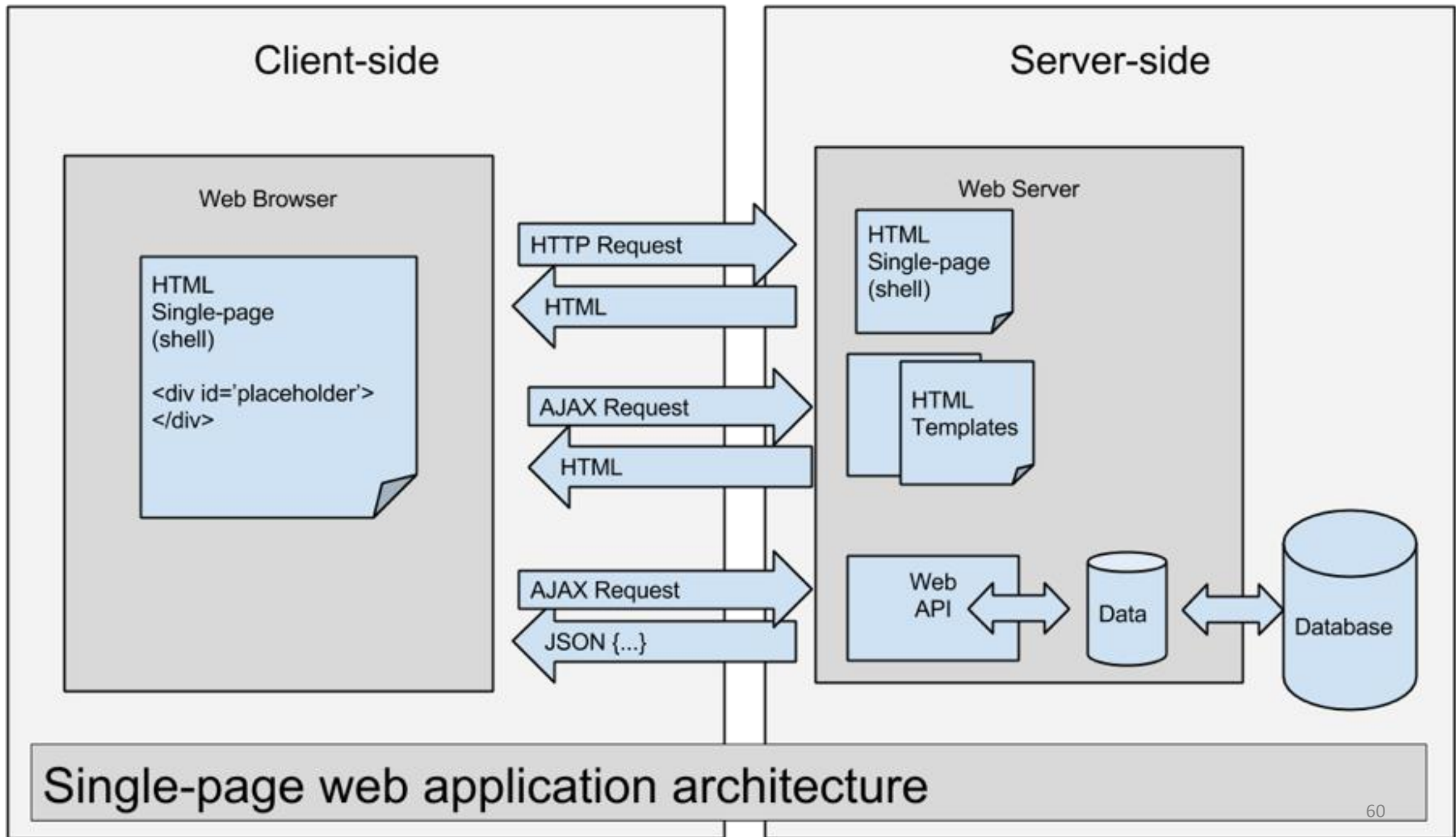
Client-side

Web Browser

HTML Page

HTML Page

HTML Page

HTTP Request

HTML

Server-side

Web Server

HTML Templates

+

Data

Database

Server-side web application architecture

Single-page web application architecture

Architecture Diagram
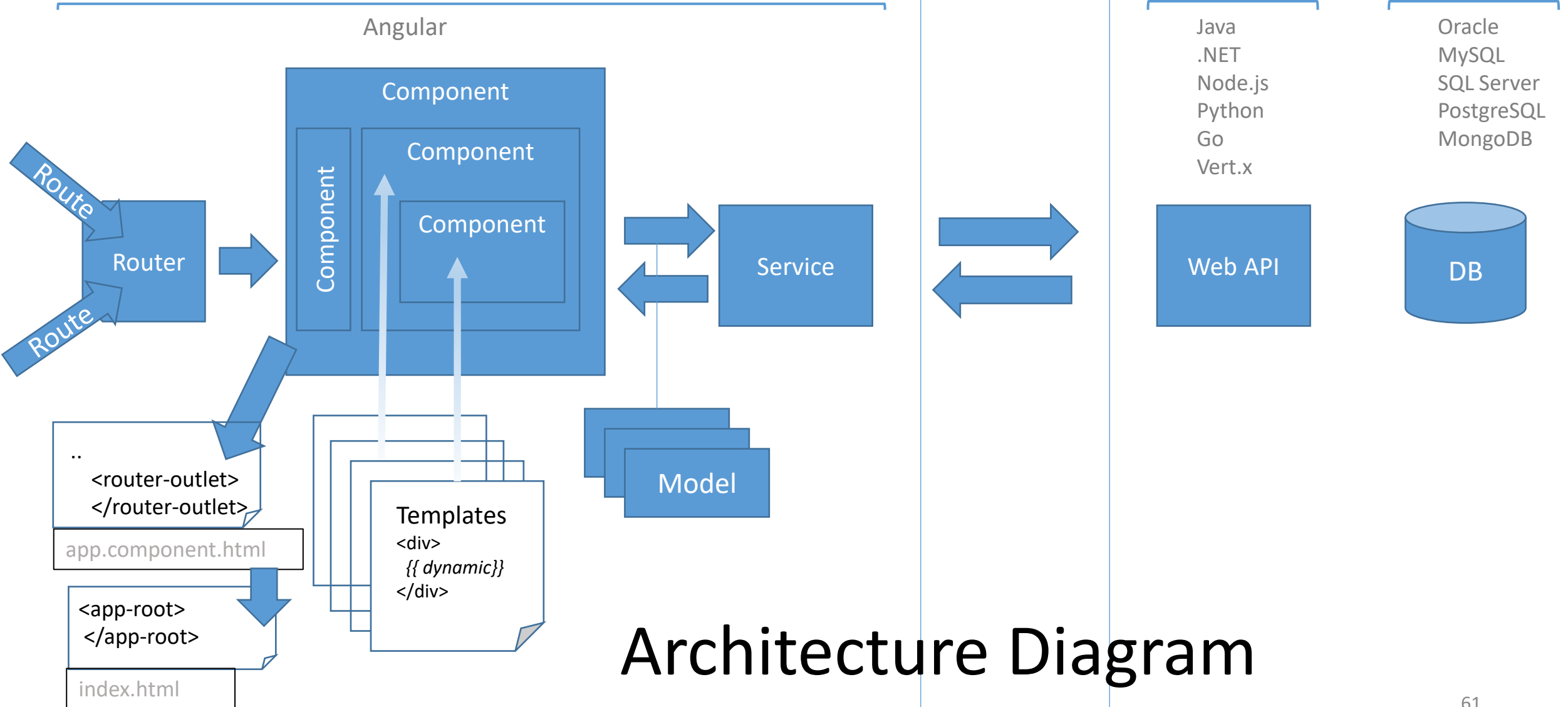
Browser Support

Browser	Supported versions
Chrome	latest
Firefox	latest
Edge	2 most recent major versions
IE	11 10 9
IE Mobile	11
Safari	2 most recent major versions
iOS	2 most recent major versions
Android	Nougat (7.0) Marshmallow (6.0) Lollipop (5.0, 5.1) KitKat (4.4)

Angular Versions

AngularJS

- 1.x

- angularjs.org

Angular

- >=2.x

- angular.io

Style Guide

- Follow the official Angular Style Guide

  - https://angular.io/docs/ts/latest/guide/style-guide.html

- Opinionated guide to Angular

  - Naming

  - Syntax

  - Conventions

  - Application structure

Angular Big Picture Review

- With a partner, write down your answers to the following questions on sheet of paper in the next 5 minutes
  - What site should I visit to get the official documentation on AngularJS?
  - What site should I visit to get the official documentation on Angular?
  - In one sentence, why are so many people adopting Angular and other Single-page application frameworks (SPA) frameworks?
  - Draw a single-page application architecture diagram.
  - Draw an Angular architecture diagram.

React & Angular Compared

React
- Facebook
- Components
- Library
  - Just the View in MVC
  - Need to include other libraries
    - React Router (Routing)
    - Axios (AJAX)
- Usually ES6 (Babel compiler)
- Create React App
  - Uses Webpack
- Redux

Angular
- Google
- Components
- Framework
  - Modular
    - Component Router
    - HttpClient
    - Forms
- Usually TypeScript (tsc compiler)
- Angular CLI
  - Uses Webpack
- Reactive Extensions for Angular (ngrx)

React vs. Angular: Key Insights

- *Angular continues to put "JS" into HTML. React puts "HTML" into JS.*
    - *–Cory House*
- Angular is a more comprehensive library while React is more of a targeted micro library.
- Because React is smaller it is:
    - Easier to understand
    - Easier to include in a project
- React is much more popular (but has existed longer)
- React is used more by design/digital/interactive agencies as well as in the Enterprise
- Angular is used more for Enterprise software particularly at larger organizations

Project Setup

using the Angular CLI (Command Line Interface)

Angular Style Guide: Naming Conventions

Symbol Name	File Name
@Component({ ... }) export class AppComponent { }	app.component.ts
@Component({ ... }) export class HeroesComponent { }	heroes.component.ts
@Component({ ... }) export class HeroListComponent { }	hero-list.component.ts
@Component({ ... }) export class HeroDetailComponent { }	hero-detail.component.ts
@NgModule({ ... }) export class AppModule	app.module.ts
@Pipe({ name: 'initCaps' }) export class InitCapsPipe implements PipeTransform { }	init-caps.pipe.ts
@Injectable() export class UserProfileService { }	user-profile.service.ts

Angular CLI: Features

- Create a new Angular application
- Run a development server with LiveReload support to preview your application during development
- Scaffolds Angular application code
- Run your application's unit tests
- Run your application's end-to-end (E2E) tests
- Build your application for deployment to production

Creating a New Project

- Install the Angular CLI

```
npm install –g @angular/cli
```

- Create a new project

```
ng new my-project --routing
cd my-project
```

Running Your Project

`ng serve --open //runs dev server at` http://localhost:4200/ and will automatically reload

or

`ng serve -o`

Angular CLI: Generating Code

- **Generate a module**

`ng generate module projects --routing`

- **Generate a component**

`ng g component projects/project-list`

g is short-hand for generate

- CLI generates files relative to the **app** folder
- Generated code follows the official **Angular Style Guide**

Customizing the Angular CLI

- Use **angular.json** to customize
- Common CLI customizations including
  - Asset Configuration
  - CSS Preprocessors
  - Global Libraries
  - Global Scripts
  - Angular Material
  - Bootstrap
  - Font Awesome
- Are documented here
  - https://github.com/angular/angular-cli/wiki/stories

Bootstrapping an Angular Application

1. Create a root component
   a. By convention named AppComponent
   b. By convention the file is named app.component.ts
   c. Add selector (custom tag) to index.html

2. Create a root module
   a. By convention named AppModule
   b. By convention the file is named app.module.ts
   c. Declare and bootstrap the root componenent
   d. Import the BrowserModule from Angular

3. Bootstrap the root module
   a) By convention the file is named main.ts
   b) platformBrowserDynamic
      i. platformBrowser means it will run in a web browser (not a native mobile app etc…)
      ii. Dynamic means the template will be compiled in the browser (not on the server)

Demos

Instructor Demonstration

Lab 1: Creating a New Project

Lab 2: Running Your Project

Lab 3: Style: Using a CSS Framework

Labs

Lab 1: Creating a New Project

Lab 2: Running Your Project

Lab 3: Style: Using a CSS Framework

Attendees Hands-On

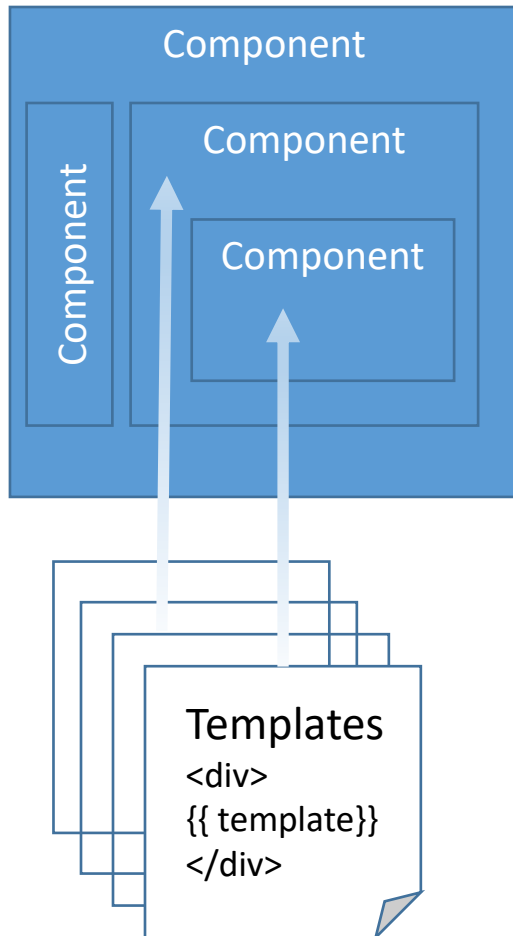# Components

Angular

CLIENT
- Front-end
- JavaScript in browser

NETWORK
(HTTP)

SERVER
- Back-end
- Web/Application server

Angular

Component

Component

Component

Component

Templates
<div>
{{ template}}
</div>

Architecture Diagram: Components

What is a Component?

- A **Component** controls a patch of screen real estate (view)

- Examples of views controlled by components
  - the shell at the application root with navigation links
  - the list of projects
  - the project form

- The view is defined using HTML in a template

- Component's application logic - what it does to support the view
  - defined in a class
  - the class interacts with the view through an API of properties and methods

AppComponent

ProjectsContainerComponent

Projects

ProjectListComponent

ProjectCardComponent

Johnson - Kutch

Fully-configurable intermediate framework. Ullam occaecati l...

Budget : $54,637

✎ Edit

ProjectFormComponent

Project Name

Wisozk Group

Project Description

Centralized interactive application.
Exercitationem nulla ut ipsum vero

Project Budget

91638

Active?

✓

Save   cancel

ProjectCardComponent

Denesik LLC

Re-contextualized dynamic moratorium. Aut nulla soluta numqu...

Budget : $29,730

✎ Edit

ProjectCardComponent

Crona Inc

Monitored explicit methodology. Rem quos maxime amet autem b...

Budget : $31,350

✎ Edit

ProjectCardComponent

Breitenberg - Mitchell

Profound upward-trending product. Neque necessitatibus quia ...

Budget : $67,030

✎ Edit

ProjectCardComponent

Christiansen LLC

Customer-focused composite implementation. Rerum ullam est v...

Budget : $98,600

✎ Edit

acme co ⌂ HOME PROJECTS

81

Left panel (browser mockup):

http://

AppComponent <app-root>

ListComponent <my-list>

ItemComponent
<my-item>

Image
Item Name
Item Description

[Edit]

ItemComponent
<my-item>

Image
Item Name
Item Description

[Edit]

ItemComponent
<my-item>

Image
Item Name
Item Description

[Edit]

FormComponent
<my-form>

Item Name

[   ]

[   ]

[Save]

ItemComponent
<my-item>

Image
Item Name
Item Description

[Edit]

ItemComponent
<my-item>

Image
Item Name
Item Description

[Edit]

Right panel (code):

index.html

```html
<html>
<body>
  <app-root>
    <my-list>                           app.component.html
      <div *ngFor="let item of items">
                                        list.component.html
        <my-item>
          <div class="list-item">       item.component.html
            <img src="..." alt="item name">
            <h3>Item Name</h3>
            <p>Item Description</p>
          </div>
        </my-item>
        <my-form>
          <div class="form">            form.component.html
            <form>
              <label>Item Name</label>
              <input type="text" name="item-name">
              <label>Item Description</label>
              <input type="text" name="item-description">
              <button>Save</button>
            </form>
          </div>
        </my-form>
      </div>
    </my-list>
  </app-root>
</body>
</html>
```
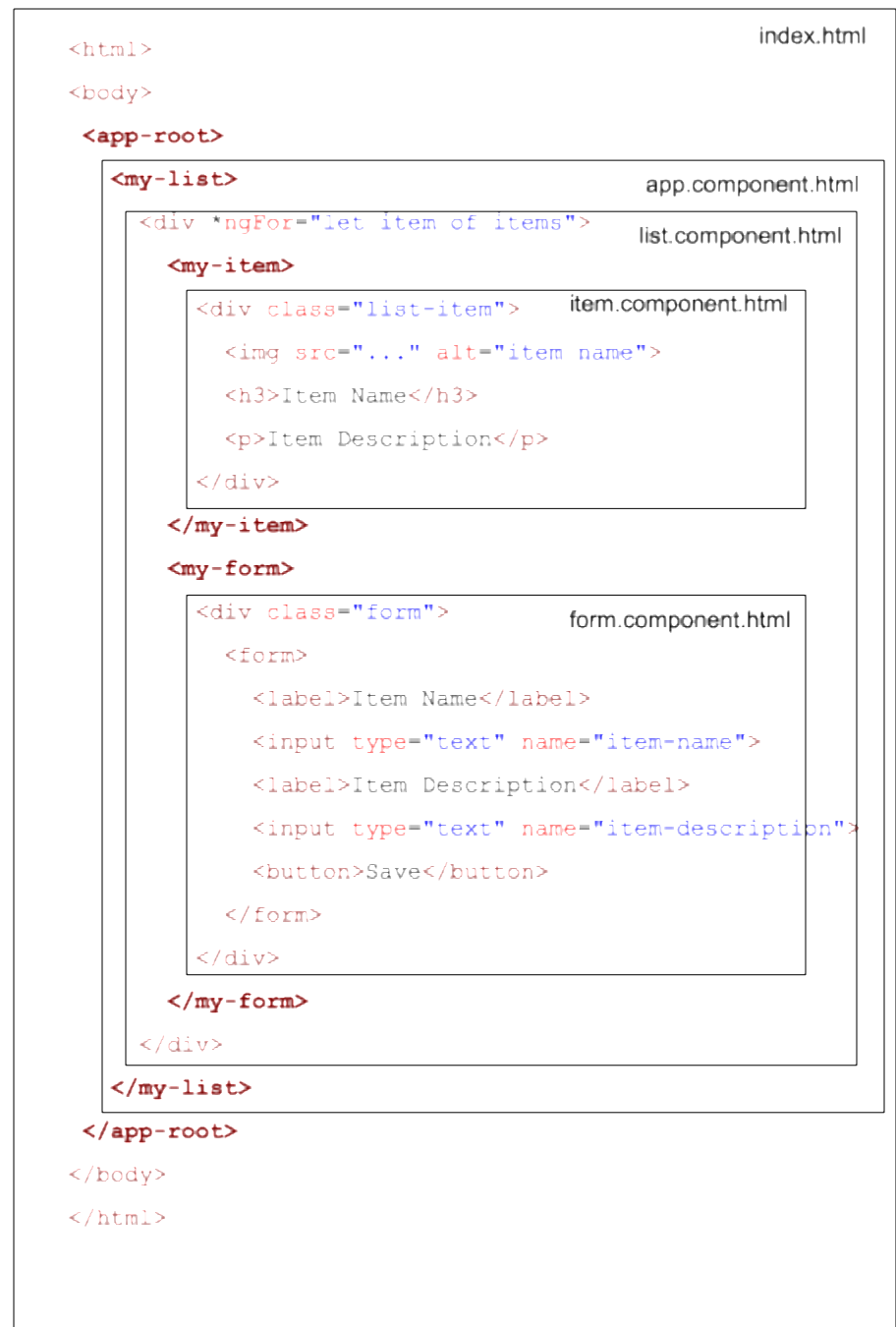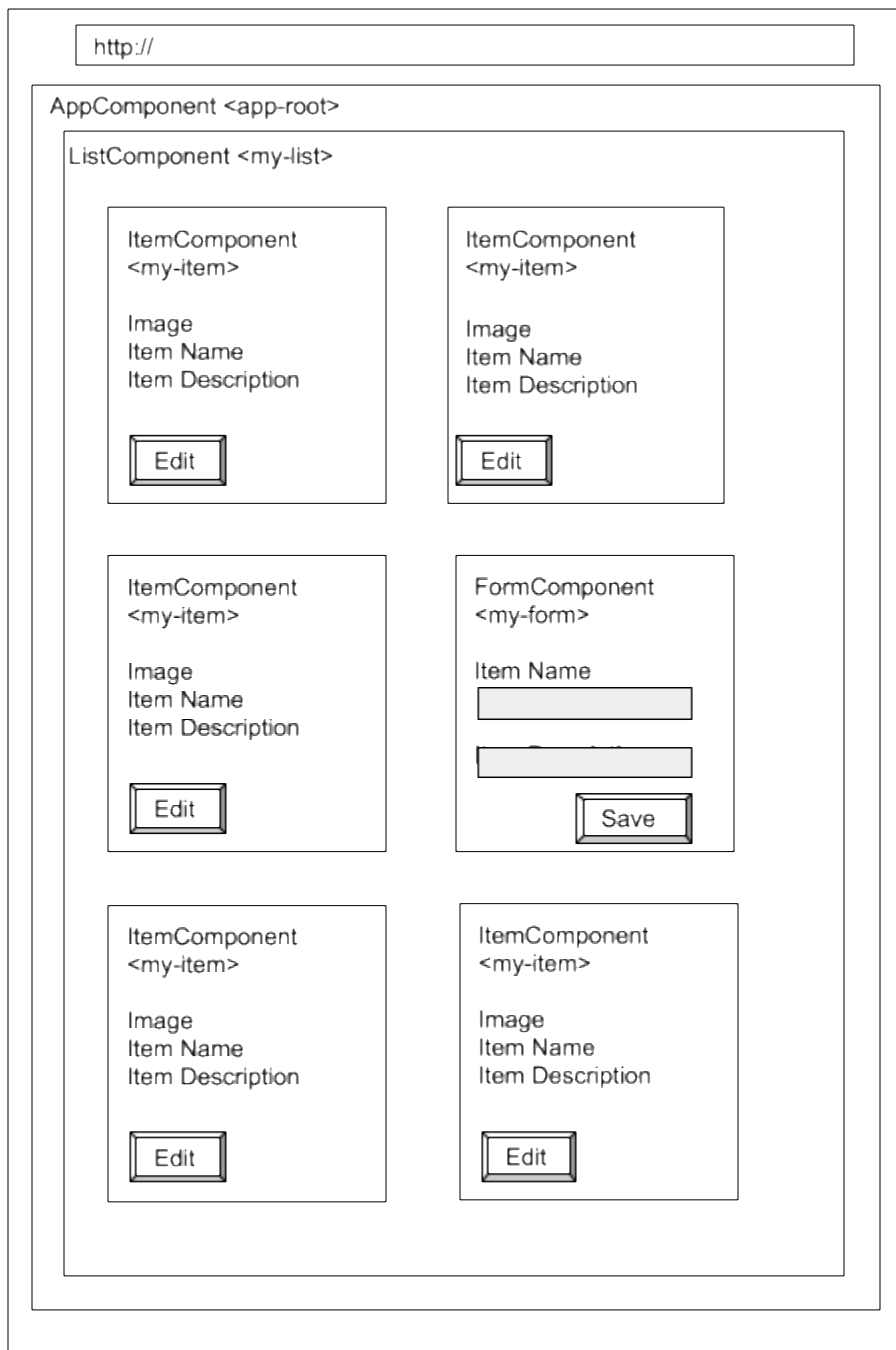
82

Demo: Component Basics

Instructor Only Demonstration

code\demos\component-first

code\demos\component-nesting

Modules

Angular

NgModule

ES Modules vs Angular Modules

JavaScript (ES) Modules

- ES6 modules represent a single file
- JavaScript modules are needed to:
  - to structure our applications (we cannot use a single file)
  - to avoid leaking code to the global namespace and thus to avoid naming collisions
  - to encapsulate code; to hide implementation details and control what gets exposed to the "outside"
  - to manage dependencies
  - to reuse code

Angular Modules

- Angular Modules are an Angular specific construct used to
- Logically group different Angular artifacts such as components, services, pipes, and directive
- Provide metadata to the Angular compiler which in turn can better "reason about our application" structure and thus introduce optimizations
- Lazy load code

Angular Module (NgModule)

- Organizes Angular code
- Logically groups different Angular framework artifacts such as components, services, pipes, and directive
- Similar to packages in Java
- Similar to namespaces in .NET
- Except Angular Modules are not organizing language constructs, but instead framework constructs

Declarations

```
@NgModule({
    declarations: [
     ProjectsContainerComponent,
     ProjectListComponent,
     ProjectCardComponent,
     ProjectFormComponent,
     ValidationErrorsComponent,
     TruncateStringPipe
     ]
})
export class ProjectsModule {}
```

If **used** in the **template** of any component listed in this module then they must be listed in **declarations**.

Angular has it's own HTML compiler. It turns Angular HTML templates into JavaScript code that generates dynamic HTML.

The compiler looks for Angular components, directives, and pipes in a template and associates them with your code.

Demo: Module Declarations

Instructor Only Demonstration

code\demos\module-declarations

Feature

- Chunk of functionality that delivers business value

- Realized by some number of user stories

- Often the same as a:
  - Table in the database
  - Entity in your domain model

Feature Modules

- Feature modules are NgModules for the purpose of organizing code

- You can organize code relevant for a specific feature

- Helps with collaboration between developers and teams, separating directives, and managing the size of the root module

- A feature module is an organizational best practice, as opposed to a concept of the core Angular API

- A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms

- Collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it share

Feature Module Example

```
@NgModule({
    imports: [
        ReactiveFormsModule,
        CommonModule,
        SharedModule,
        ProjectsRoutingModule
    ],
    declarations: [
        ProjectsContainerComponent,
        ProjectListComponent,
        ProjectCardComponent,
        ProjectFormComponent,
        ValidationErrorsComponent
    ],
    providers: [ProjectService]
})
export class ProjectsModule {}
```

Imports are always modules and are always named with a Module suffix.

These modules can be parts of the Angular framework, your app's reusable code, or your app's other features or routing modules.

If used in the template of any component listed in this module then they must be listed in declarations.

Services can be registered in providers.

Application Structure

- LIFT
  - **L**ocate code quickly
  - **I**dentify the code at a glance
  - Keep the **F**lattest structure you can
  - **T**ry to be DRY
    - Avoid being so DRY that you sacrifice readability
- *Folders-by-feature*
  - Do create folders named for the feature area they represent.

```
<project root>
  src
    app
      core
        core.module.ts
        exception.service.ts|spec.ts
        user-profile.service.ts|spec.ts
      heroes
        hero
          hero.component.ts|html|css|spec.ts
        hero-list
          hero-list.component.ts|html|css|spec.ts
        shared
          hero-button.component.ts|html|css|spec.ts
          hero.model.ts
          hero.service.ts|spec.ts
        heroes.component.ts|html|css|spec.ts
        heroes.module.ts
        heroes-routing.module.ts
      shared
        shared.module.ts
        init-caps.pipe.ts|spec.ts
        text-filter.component.ts|spec.ts
        text-filter.service.ts|spec.ts
      villains
        villain
          ...
        villain-list
          ...
        shared
          ...
        villains.component.ts|html|css|spec.ts
        villains.module.ts
        villains-routing.module.ts
      app.component.ts|html|css|spec.ts
      app.module.ts
      app-routing.module.ts
    main.ts
    index.html
    ...
  node_modules/...
  ...
```

Demo: Module Imports & Exports

Instructor Only Demonstration

code\demos\module-imports-exports

Lab

Lab 4: Your First Component

Attendees Hands-On

Root Module vs Feature Modules

- Every Angular app has at least one module, the *root module*, conventionally named AppModule.

- While the *root module* may be the only module in a small application, most apps have many more *feature modules*.
  - A feature module is a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

- An Angular module, whether a *root* or *feature*, is a class with an @NgModule decorator.

Models
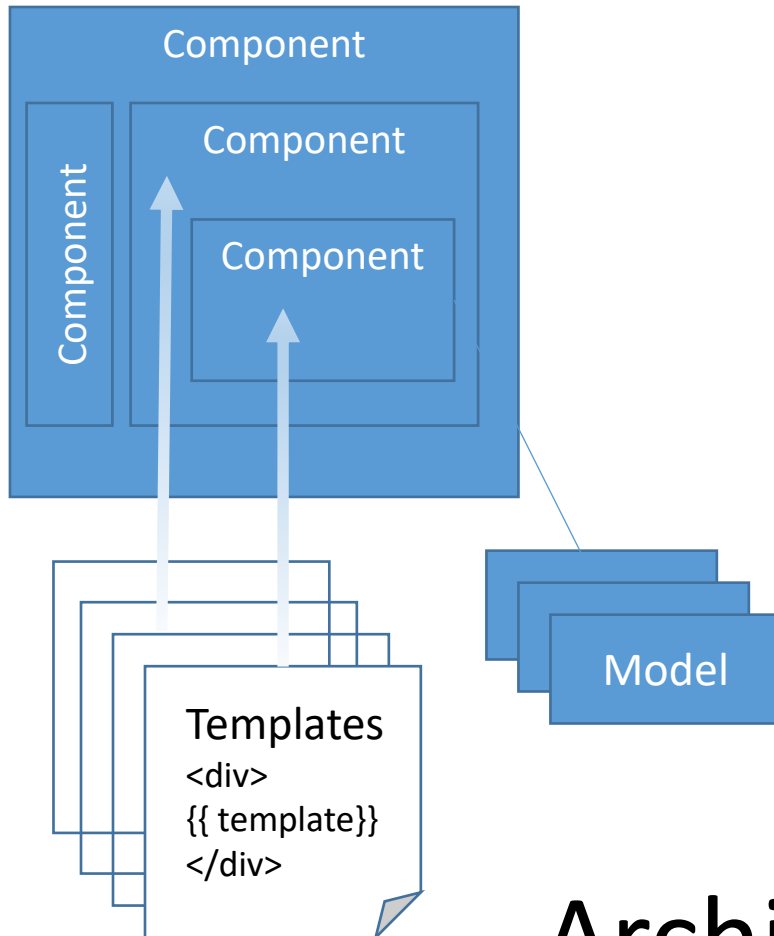
Architecture Diagram: Models

Models

are just classes

```typescript
export class Project {
    constructor(
    public id: number,
    public name: string,
    public description: string,
    public imageUrl: string,
    public contractTypeId: number,
    public contractSignedOn: Date,
    public budget: number,
    public isActive: boolean,
    ) {}
}
```

Models

constructor overloads

```typescript
export class Project {
    id: number;
    name: string;
    description: string;
    isActive: boolean;
    contractSignedOn: Date;
    budget: number;

    constructor(obj?: any) {
        this.id = (obj && obj.id) || null;
        this.name = (obj && obj.name) || null;
        this.description = (obj && obj.description) || null;
        this.contractTypeId = (obj && obj.contractTypId) || null;
        this.isActive = (obj && obj.isActive) || false;
        this.contractSignedOn = (obj && obj.contractSignedOn) || new Date();
        this.budget = (obj && obj.budget) || 0;
    }
}
let project = new Project({ name: 'Acme Website Redesign', budget: 30000 });
```

JSON Pipe

```
@Component({
    selector: 'json-pipe',
    template: `<div>

    <p>Without JSON pipe:</p>
    <pre>{{object}}</pre>

    <p>With JSON pipe (no pre tag):</p>
    <p>{{object | json}}</p>

    <p>With JSON pipe (and pre tag):</p>
    <pre>{{object | json}}</pre>

    </div>`,
})
export class JsonPipeComponent {
...
}
```

Angular CLI: Generate Class

```
ng g class projects/shared/project.model
```

ngFor

```
@Component({
    selector: 'app-ng-for-demo',
    template: `
        <ul>
            <li *ngFor="let fruit of fruits">
                {{fruit}}
            </li>
        </ul>
    `,
})
export class NgForComponent {
        fruits = ['Apple', 'Orange', 'Plum'];
}
```

Start with *

of
not
in

Demo: ngFor

Instructor Only Demonstration

code\demos\ngFor

Data Binding

Angular

Data Binding
Four forms (types)

{{value}}

[property] = "value"

(event) = "handler"

DOM

COMPONENT

[(ng-model)] = "property"
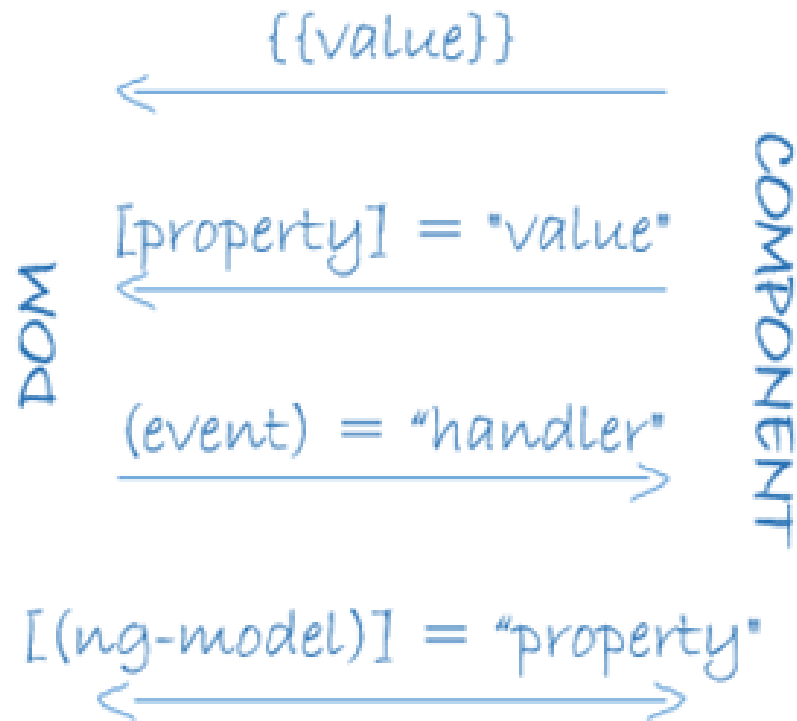
Image from angular.io

- **Interpolation**

- Property binding

- Event binding

- Two-way data binding

Interpolation

```
@Component({
    selector: 'app-root',
    template: `
    <h2>{{image.name}}</h2>
    <p>{{image.path}} </p>
    `,
    styles: []
})
export class AppComponent {
    image = {
        path: '../assets/angular_solidBlack.png',
        name: 'Angular Logo'
    };
}
```

Data Binding
Four forms (types)

{{value}}

[property] = "value"

DOM

COMPONENT

(event) = "handler"

[(ng-model)] = "property"
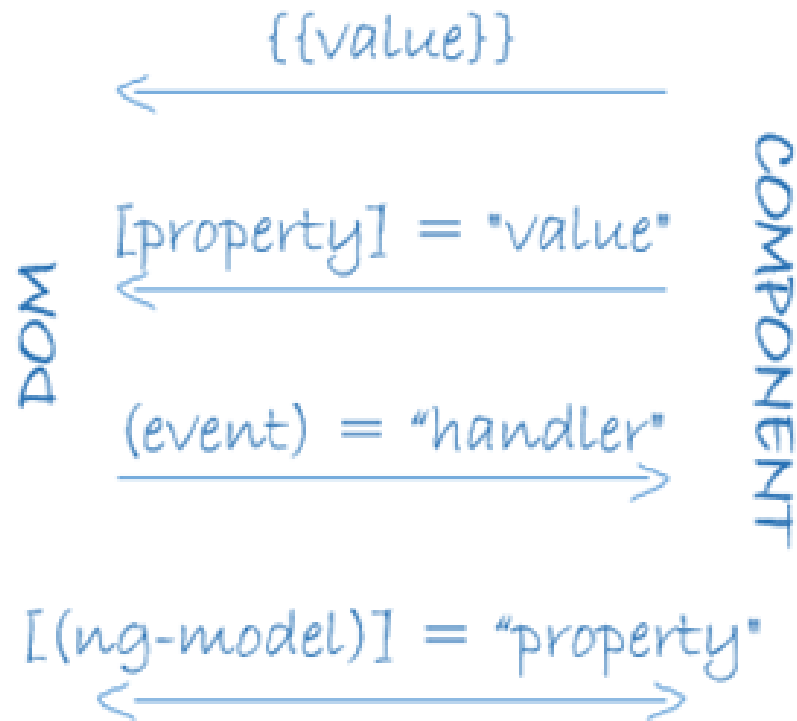
Image from angular.io

- Interpolation

- **Property binding**

- Event binding

- Two-way data binding

Property Binding

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

@Component({
    selector: 'app-root',
    template: `
    <img [src]="image.path" [title]="image.name"
        [alt]="image.name">
    `,
})
export class AppComponent {
    image = {
            path: '../../assets/angularlogo.png',
            name: 'Angular Logo',
    };
}
```

Demo: Data Binding

Interpolation & Property Binding

Instructor Only Demonstration

code\demos\interpolation

code\demos\property-binding

Input Property

- @Input
  - Decorator that marks a class field as an input property
- Property Binding to a Component Property

```
@Component({
    selector: 'app-root',
    template: `
    <app-fruit-list [fruits]="data"></app-fruit-list>
    `,
    styles: []
})
export class AppComponent {
  data: string[] = ['Apple', 'Orange', 'Plum'];
}
```

Demo: Input Property

Instructor Only Demonstration

code\demos\input-property

Labs

Lab 5: Creating Data Structures (Models)

Lab 6: Passing Data into a Component

Lab 7: Looping Over Data

Attendees Hands-On

Data Binding
Four forms (types)

Image from angular.io

- Interpolation

- Property binding

- **Event binding**

- Two-way data binding

114

Event Binding

```
@Component({
    selector: 'app-event-binding-demo',
    template: `
    <a href="/event-binding" (click)="onClick($event)">Click Me!</a>
    <p [innerText]="message"></p>
    `
})
export class EventBindingComponent {
    message = '';

    onClick(event) {
        event.preventDefault();
        this.message = 'clicked';
    }
}
```

$event is template variable available in Angular

Can use any standard browser event.

https://developer.mozilla.org/en-US/docs/Web/Events#Standard_events

Prevents the default browser behavior for that element.

https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault

Demo: Event Binding

Instructor Only Demonstration

code\demos\event-binding

Pipes

Angular

CLIENT
- Front-end
- JavaScript in browser
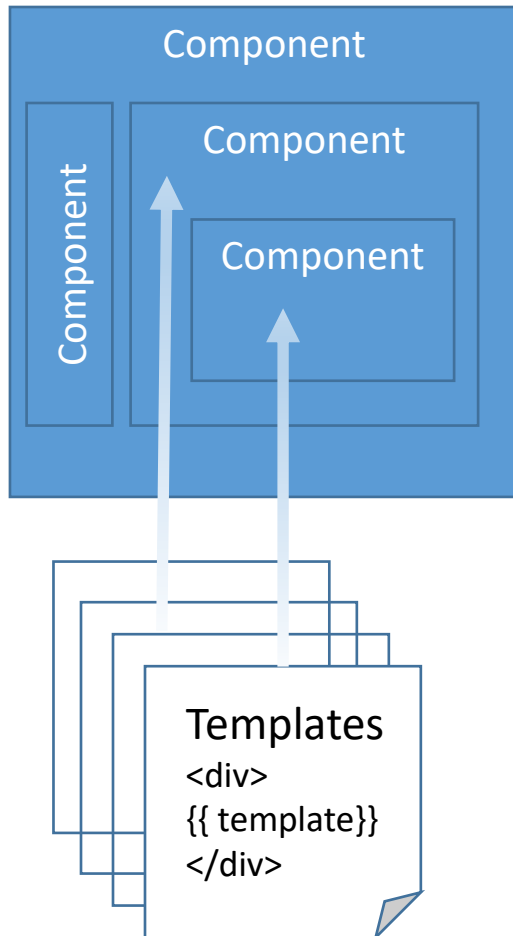
NETWORK
(HTTP)

SERVER
- Back-end
- Web/Application server

Angular

Component
Component
Component
Component

Templates
<div>
{{ template}}
</div>

Architecture Diagram: Templates

What are Pipes?

- A pipe takes in data as input and transforms it to a desired output
- Commonly used to format data
- Get their name from the pipe operator "|"
- Can format various types: String, Number, Data, Array

Using Pipes

- Pipes can be used
  - In HTML templates
  - In JavaScript (often in components)
- Pipes can be
  - Built-in to Angular
  - Custom pipes created by application developers
    - Custom pipes can be used to implement any type of formatting

Changes from AngularJS

- Filters in AngularJS are Pipes in Angular
- The "filter" filter is not built-in Angular
  - Was used to filter out arrays based on a search term
- The "orderBy" filter is not built-in to Angular
  - Was used to sort array elements for display
- "filter" and "orderBy"
  - Prone to breaking if code was minified
  - Could lead to performance issues
  - Replacements can be implemented as a custom filter

Using a Built-in Pipe

`<h4>{{project.name | uppercase}}</h4>`

Common Built-In Pipes

Class	Name
PercentPipe	percent
UpperCasePipe	uppercase
LowerCasePipe	lowercase
TitleCasePipe	titlecase
DatePipe	date
DecimalPipe	decimal
CurrencyPipe	currency

Pipe Syntax

- Syntax

{{data-expression | pipe-name [:pipe-parameter: pipe-parameter-2…]}}

- Example

releaseDate: Date(1977, 5, 25);

{{releaseDate| date: ”M-dd-yyyy”}}

Chaining Pipes

- Chained pipes are executed from left to right

    {{ releaseDate| date | uppercase}}

- The *releaseDate* property is converted to a date string

- The date string is then converted to uppercase

Decimal Pipe

- Takes a number as input
- And a *digitInfo* string defining the number format {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
- Example
  - Usage: {{47.243 | number: "3.2-4"}}
  - Output: 047.243

CurrencyPipe

- Use [currency](#) to format a number as currency.
- Takes a number and up to three parameters
  - *currencyCode* is the [ISO 4217](#) currency code, such as USD for the US dollar and EUR for the euro.
  - *display* indicates whether to use the currency symbol or code.
    - code: use code (e.g. USD).
    - symbol(default): use symbol (e.g. $).
    - symbol-narrow: some countries have two symbols for their currency, one regular and one narrow (e.g. the canadian dollar CAD has the symbol CA$ and the symbol-narrow $).
    - boolean (deprecated from v5): true for symbol and false for code If there is no narrow symbol for the chosen currency, the regular symbol will be used.
  - *digitInfo* defining the number format:
    - "{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}"
  - Example:
    - Usage: {{47.341| currency: 'USD': '2.1-2'}}
    - Output: $47.34

Demo: Input Property

Instructor Only Demonstration

code\demos\pipes

Using Pipes in JavaScript

- Syntax
  - pipeClass.transform(value, param1, param2, etc);

- In the module:
  - **import** { DatePipe } **from '@angular/common'**;
  - @NgModule({
    providers: […, DatePipe]
    })

- In the component:
  constructor(private datePipe: DatePipe){}

  releaseDate: Date = new Date(1975, 5, 25);
  releaseDateFmt: string = this.datePipe.transform(this.releaseDate, "M-dd-yyyy")

How to Structure an Application

1. Components
   - If a component gets too complex split it into smaller components
2. After you create more components, more questions arise
   - What types of components are there?
   - How should components interact?
   - Should I inject services into any component?
   - How do I make my components reusable across views?

Component Architecture

Smart/Container Components

- Are concerned with *how things work*

- *Sets data* into child component input properties

- *Receives events* by subscribing to children

- *Loads and modifies data* via calls to an API

- Also know as *container* components or *controller* components

Presentation Components

- Are concerned with *how things look*

- *Receive data* via input properties from parent

- *Send events* with information to their parent

- Don't specify how the data is loaded or changed

- Also know as *pure* components or *dumb* components

When to Create Another Component

- Is it possible for your code chunk to be reused?
    - If yes, construction of a new component seems like a great idea.
    - Even if the reuse is within a single component.

- Is your code quite complex?
    - If yes maybe its good idea to split in separate components in order to make your code more readable and maintainable.

Custom Events in a Component

- ## @Output
  - Decorator that marks a class property as sending a custom output event
- ## EventEmitter
  - Class used in directives and components to emit custom events synchronously or asynchronously, and register handlers for those events by subscribing to an instance

Demo: Output Events

Instructor Only Demonstration

code\demos\output-events

Component Styles

- Angular applications are styled with regular CSS
- Angular has the ability to bundle *component styles* with our components
  - enables a more modular design than regular stylesheets

Component Styles

demos/components/styling-external.ts

External Styles

```
@Component({
    selector: 'styling-external',
    template: '<h1>Styling Components: External</h1>',
    styleUrls: ['./styling-external.css'],
})
export class StylingExternalComponent {}
```

Reference external style sheet.

demos/components/styling-external.css

```
h1 {
color: rgb(255, 165, 0);
}
```

Use CSS to style your template.

By default these styles will be local to this component and not affect the rest of the page (ViewEncapsulation.Emulated).

Component Styles

View Encapsulation

- Component CSS styles are *encapsulated* into the component's own view and do not affect the rest of the application (when using the default)

- We can control how this encapsulation happens on a *per component* basis by setting the *view encapsulation mode* in the component metadata

- There are three modes to choose from
  - Native
  - Emulated
  - None

Component Styles

Emulated View Encapsulation

Demo: Input Property

Instructor Only Demonstration

code\demos\component-styles

Labs

Directives

Built-In to Angular

Architecture Diagram: Templates

What is a Directive?

- Angular transforms the DOM according to the instructions given by **directives**

- A directive is a class with directive metadata.
  - In TypeScript, we apply the @Directive decorator to attach metadata to the class

Kinds of Directives

- Component
  - A **component** is a *directive-with-a-template*
- Structural
  - **Structural** directives alter layout by adding, removing, and replacing elements in DOM
- Attribute
  - **Attribute** directives alter the appearance or behavior of an existing element

Structural Directives

- **Structural** directives alter layout by adding, removing, and replacing elements in DOM
    - ngIf
    - ngFor
    - ngSwitch

ngIf

- Takes a boolean and makes an entire chunk of DOM appear or disappear

```
<p *ngIf="condition">
    condition is true and ngIf is true.
</p>

<p *ngIf="!condition">
    condition is false and ngIf is false.
</p>
```

Demo: ngIf

Instructor Only Demonstration

code\demos\ngIf

Labs

Lab 13: Hiding and Showing Components

Lab 14: Preventing a Page Refresh

Lab 15: More Component Communication *(optional, do only if you are finishing labs early and want extra practice, similar to Lab 12)*

Attendees Hands-On Together

Remove or Hide

ngIf or hidden

```html
<p *ngIf="condition">
  Element to be added or removed
</p>
<button class="btn btn-warning" (click)="toggleIf()" >add | remove</button>
```

```html
<p [hidden]="!isVisible" >
 Element to show or hide using CSS
</p>
<button class="btn btn-warning"(click)="toggleVisibility()">
 show | hide
</button>
```

Hide or Remove

hidden or ngIf

Hide: using [hidden]

The component's behavior continues

Remains attached to its DOM element

Continues to listen to events

Angular keeps checking for changes that could affect data bindings

The component — and all of its descendent components — tie up resources

Performance and memory burden can be substantial

Showing again is quick

Remove: using ngIf

The component's behavior stops

DOM element is removed

Stops listening to events

Angular stops checking for changes in data bindings

The component — and all of its descendent components — are cleaned up

Performance and memory burden are signifantly reduced

Showing again can be slow

Remove or Hide Heuristic

hidden or ngIf

- In general, it is best to **use ngIf** to remove unwanted components rather than hide them

- The *more complicated* the element is, the *more likely* this will be the *right choice*

Understand How Structural Directives Work

```
<!-- Examples (A) and (B) are the same -->

<!-- (A) *ngIf paragraph -->
<p *ngIf="condition">
    Our heroes are true!
</p>

<!-- (B) [ngIf] with template -->
<ng-template [ngIf]="condition">
    <p>
        Our heroes are true!
    </p>
</ng-template>
```

NgSwitch

```
@Component({
  selector: 'my-app',
  template: `
  <div class="container">
  <button (click)="value=1">select - 1</button>
  <button (click)="value=2">select - 2</button>
  <button (click)="value=3">select - 3</button>
  <h5>You selected : {{value}}</h5>

  <hr>
  <div [ngSwitch]="value">

      <div *ngSwitchCase="1">1. Template - <b>{{value}}</b> </div>
      <div *ngSwitchCase="2">2. Template - <b>{{value}}</b> </div>
      <div *ngSwitchCase="3">3. Template - <b>{{value}}</b> </div>
      <div *ngSwitchDefault>Default Template</div>

  </div>
  </div>
  `
})
export class AppComponent {
        value: number;
}
```

Demo: ngSwitch

Instructor Only Demonstration

code\demos\ngSwitch

Forms

Reactive Forms

Benefits of Forms

- Forms are the mainstay of business applications
  - Login
  - Place an Order
  - Book a Flight
  - Schedule a Meeting

Form Strategies

Template-driven

- AngularJS style

- FormsModule

- Use the core ng prefix

- Declarative in the template

Model-driven (new)

- Reactive programming

- ReactiveFormsModule

- Use the form prefix

- Configured in component code

- Enables dynamic forms

- Facilitates unit testing
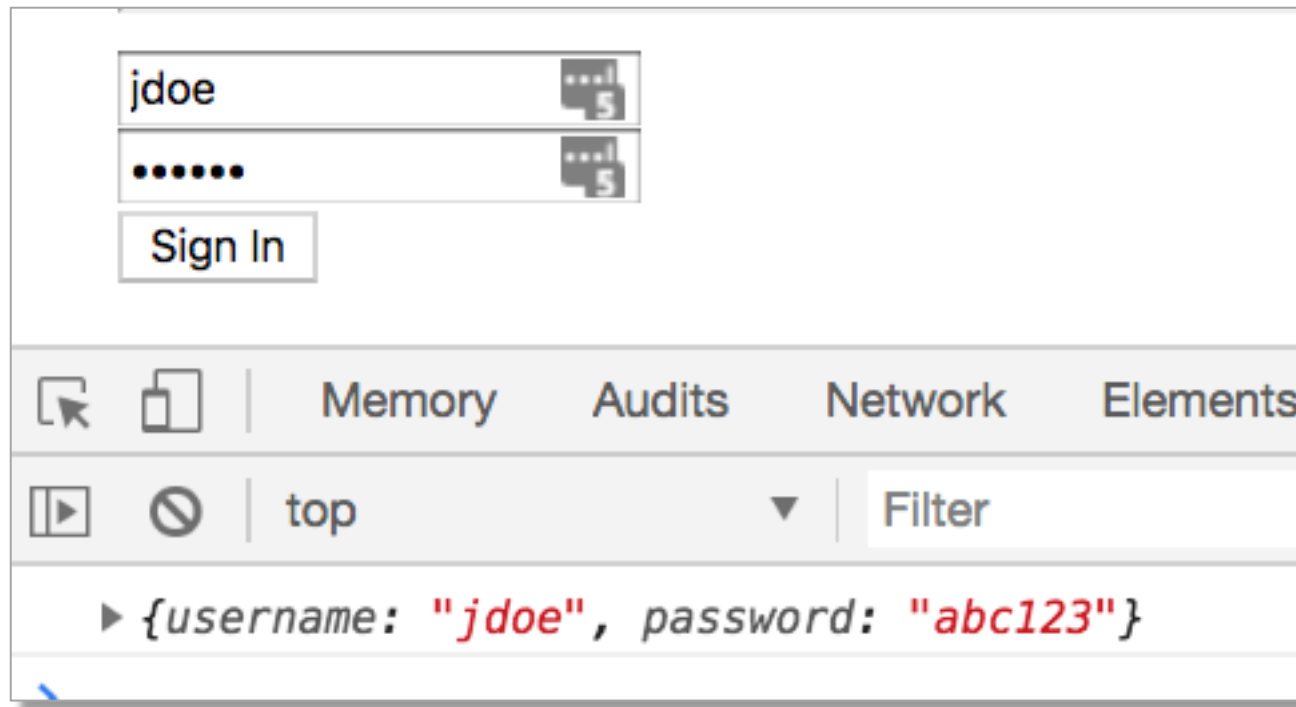
Demo: Reactive Forms Binding

Instructor Only Demonstration

code\demos\reactive-forms-binding

Labs

Lab 16: Forms | Binding

Demo: Reactive Forms Validation

Instructor Only Demonstration

code\demos\reactive-forms-validation

Demo: ngClass

Instructor Only Demonstration

code\demos\ngClass

Labs

Lab 16: Forms | Binding

Lab 17: Forms | Saving

Lab 18: Forms | Validation

Attendees Hands-On

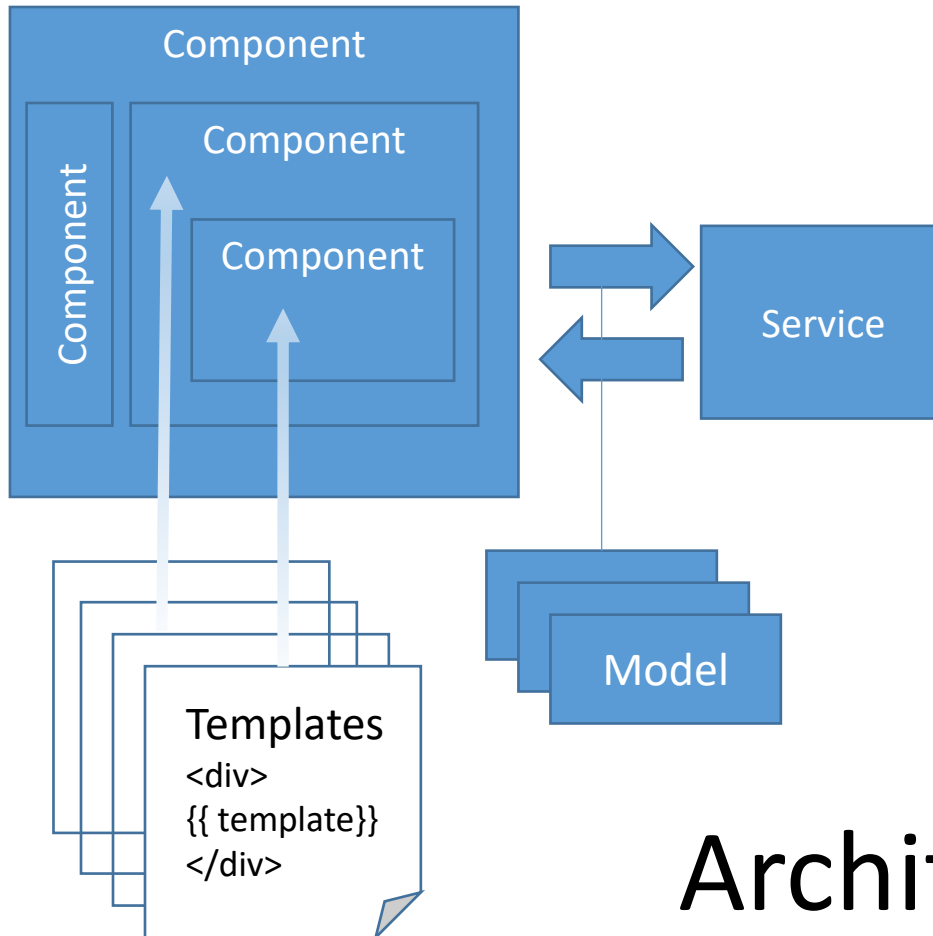Lab 19: Forms | Refactor (Instructor Walkthrough)

Services

Angular

CLIENT
• Front-end
• JavaScript in browser

NETWORK
(HTTP)

SERVER
• Back-end
• Web/Application server

Angular

Component

Component

Component

Component

Service

Model

Templates
<div>
{{ template}}
</div>

Architecture Diagram:
Services

166

What is a Service?

- *Service* is a broad category, almost anything can be a service
- A service is typically a class with a narrow, well-defined purpose
- It should do something specific and do it well

Service Compared to Component

- By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

- Ideally, a component's job is to enable the user experience and nothing more.

- A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an *injectable service class*, you make those tasks available to any component.

Data Service
project.service.ts
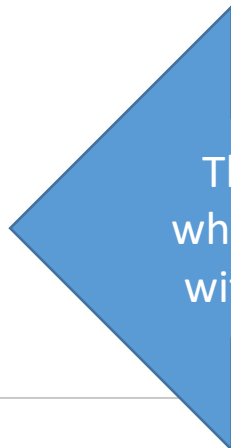
```typescript
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Project } from './project.model';
import { PROJECTS } from './mock-projects';

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

  constructor() { }

  list(): Observable<Project[]> {
    return of(PROJECTS);
  }
}
```

Provides this service in the root module's injector.
We cover dependency injection and injectors in the next section.

The RxJS **of** method creates an **Observable** which (like a Promise) allows us to easily work with Asynchronous operations like AJAX calls

Using a Data Service in a Component

projects-conainter.component.ts

```typescript
export class ProjectsContainerComponent implements OnInit {
    projects: Project[] = PROJECTS;

    constructor(private projectService: ProjectService) {}

    ngOnInit() {
        this.projectService.list()
        .subscribe(data => {
                this.projects = data;
        });
    }

}
```

OnInit is an **interface** that has one method **ngOnInit** which is a component lifecyle event

Tell Angular you need an instance of the service and the framework will **Inject** it into the constructor.

The **list** method returns an RxJS **Observable.** We **subscribe** to the **Observable** and when it has data it will call the arrow function we passed as an argument.

Demo: Services

Instructor Only Demonstration

code\demos\services

Labs

Lab 20: Services

Attendees Hands-On

Dependency Injection

Angular

Dependency Injection
explained

- Imagine you are not allowed to use the new keyword and create instances of other objects (dependencies) you need when writing code

- Dependency Injection is a practice where objects are designed in a manner where they receive instances of the objects from other pieces of code, instead of constructing them internally

- This means that any object implementing the interface which is required by the object can be substituted in without changing the code, which simplifies testing, and improves decoupling

Dependency Injection
example

```
//no DI
class CustomersComponent{
    private customerService = new CustomerDataAccessService ();
}
```

```
//using DI
class CustomersComponent{
 private customerService : CustomerDataAccessService;

    constructor(customerService : CustomerDataAccessService){
        this.customerService = customerService;
    }
}
```

Dependency Injection Frameworks

- Java
  - Spring, Guice

- .NET
  - Ninject, Structure Map, Unity, Spring.NET, .NET Core (built-in)

- Angular
  - Dependency Injection is built-in
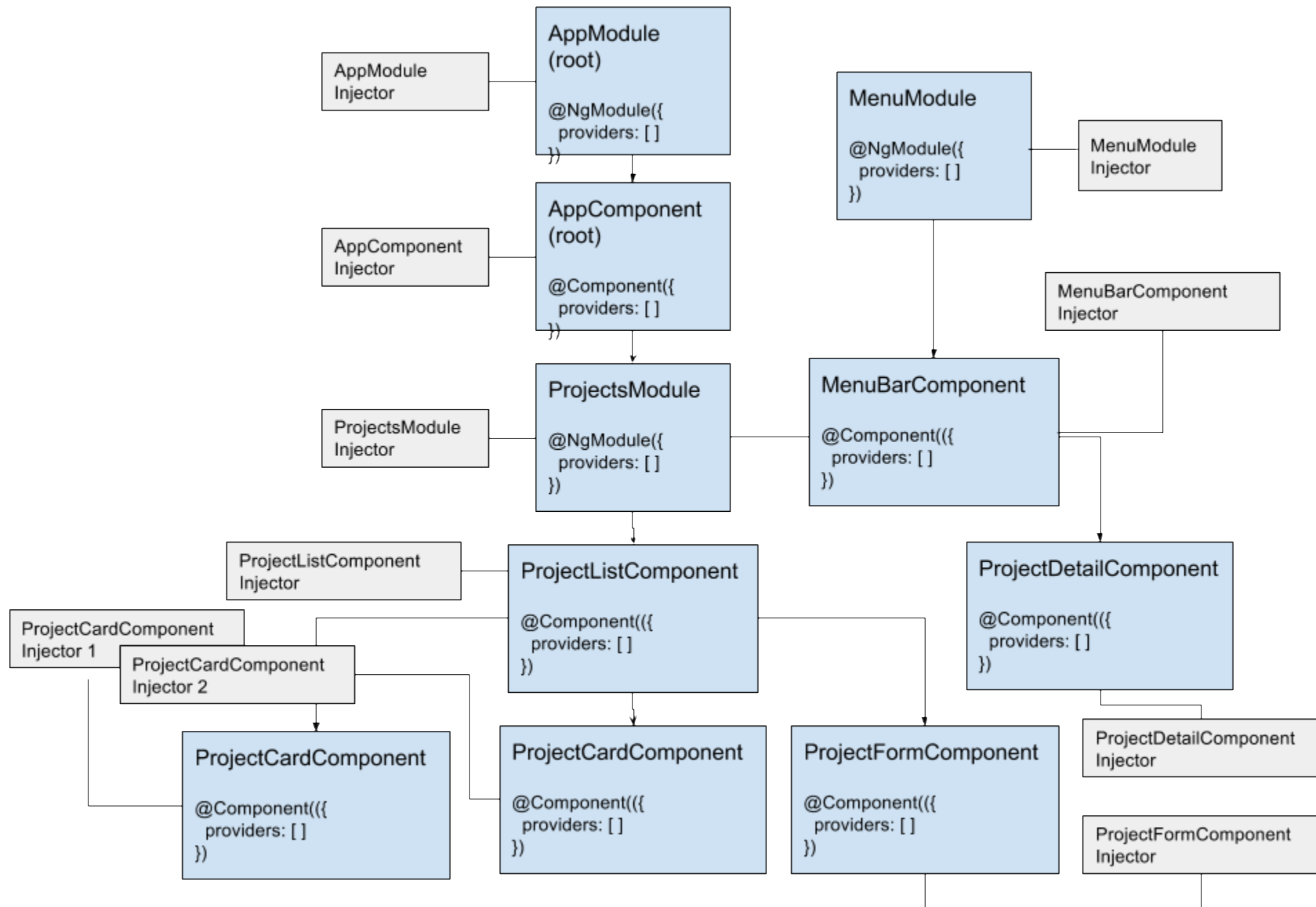
Hierarchical Dependency Injectors

Providing Services in the Root Module Injector
project.service.ts

```typescript
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';

import { Project } from './project.model';
import { PROJECTS } from './mock-projects';



@Injectable({
  providedIn: 'root'
})
export class ProjectService {
  list(): Observable<Project[]> {
    return of(PROJECTS);
  }
}
```

Provides this service in the **root** injector (**Singleton**)

RxJS **of** method creates an Observable

Providing Services in a Feature Module Injector
project.service.ts

```typescript
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';


import { Project } from './project.model';
import { PROJECTS } from './mock-projects';



@Injectable({
  providedIn: ProjectsModule
})
export class ProjectService {
  list(): Observable<Project[]> {
    return of(PROJECTS);
  }
}
```

Provides this service in the **ProjectsModule** injector

Service Registration Best Practices

- Provide services in the Service itself (providedIn)
- Set **providedIn** to
  - **root** if you want the service to be a **Singleton**
  - **Feature Module** (*ProjectsModule*) if you want the service to be **lazy-loaded**
    - **Lazy Loading** requires additional steps which are covered in the Routing section of the course
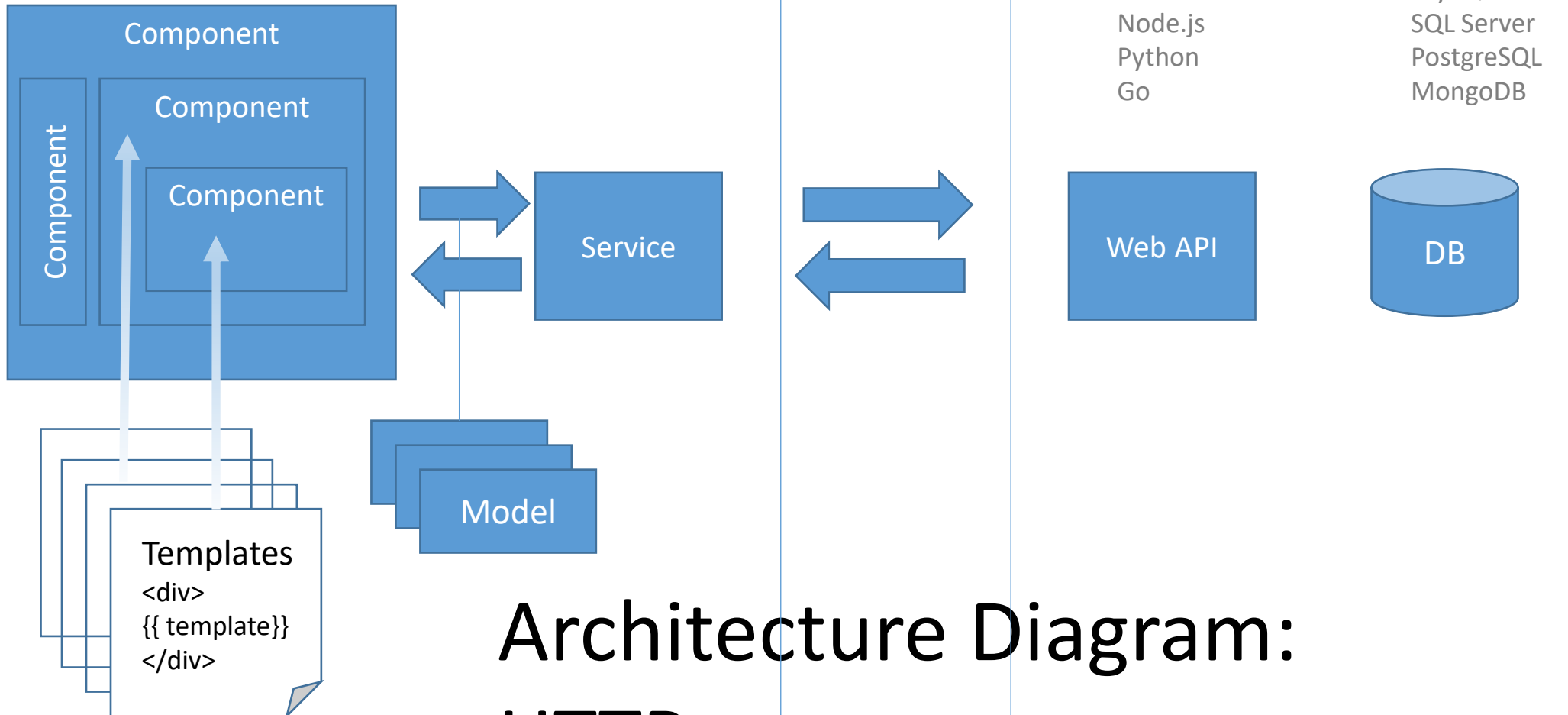
Http

Angular

Architecture Diagram: HTTP

Labs

Lab 21: Setup Backend REST API (attendees with instructor)

Rest Review (instructor only demonstration) (optional)

HttpClient Overview

- Import HttpClientModule
- HttpClient in Services
- Observables in Components

Import HttpClientModule

```typescript
import { HttpClientModule } from '@angular/common/http';

@NgModule({
    declarations: [AppComponent],
    imports: [BrowserModule, AppRoutingModule,
                    ProjectsModule, HttpClientModule],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

HttpClient in Services

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

import { environment } from '../../../environments/environment';


...

export class ProjectService {
 private projectsUrl = environment.backendUrl + '/projects/';


 constructor(private http: HttpClient) {}


 list(): Observable<Project[]> {
  return this.http.get<Project[]>(this.projectsUrl);
  }
}
```

Observables in Components

```
...
export class ProjectsContainerComponent implements OnInit {
 projects: Project[];
 errorMessage: string;

 constructor(private projectService: ProjectService) {}
 ngOnInit() {
  this.projectService.list().subscribe(
    data => {
     this.projects = data;
    }
   );
 }
 ...
}
```

Labs

Lab 22: HTTP GET

Demos: Http Get

Instructor Only Demonstration

code\demos\http-get

Error Handling in Services

```
import { Observable, of, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';

...
export class ProjectService {
...
  list(): Observable<Project[]> {

    return this.http.get<Project[]>(this.projectsUrl).pipe(
      catchError((error: HttpErrorResponse) => {
        console.log(error);
        return throwError('An error occurred loading the projects.');
      })
    );

  }
}
```

Error Handling in Components

```
...
export class ProjectsContainerComponent implements OnInit {
 projects: Project[];
 errorMessage: string;

 constructor(private projectService: ProjectService) {}
 ngOnInit() {
  this.projectService.list().subscribe(
    data => {
     this.projects = data;
    },
    error => {
     this.errorMessage = error;
    }
   );
 }
 ...
}
```

Demos: Http Error Handling

Instructor Only Demonstration

code\demos\http-error-handling

Labs

Lab 23: HTTP Error Handling

HttpClient PUT

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

const httpOptions = {
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};

export class ProjectService {
...
  put(project: Project): Observable<Project> {
    const url = this.projectsUrl + project.id;
    return this.http.put<Project>(url, project, httpOptions).pipe(
      catchError((error: HttpErrorResponse) => {
        console.log(error);
        return throwError('An error occurred updating the projects.');
      })
    );
  }
}
```

Labs

Lab 24: HTTP PUT

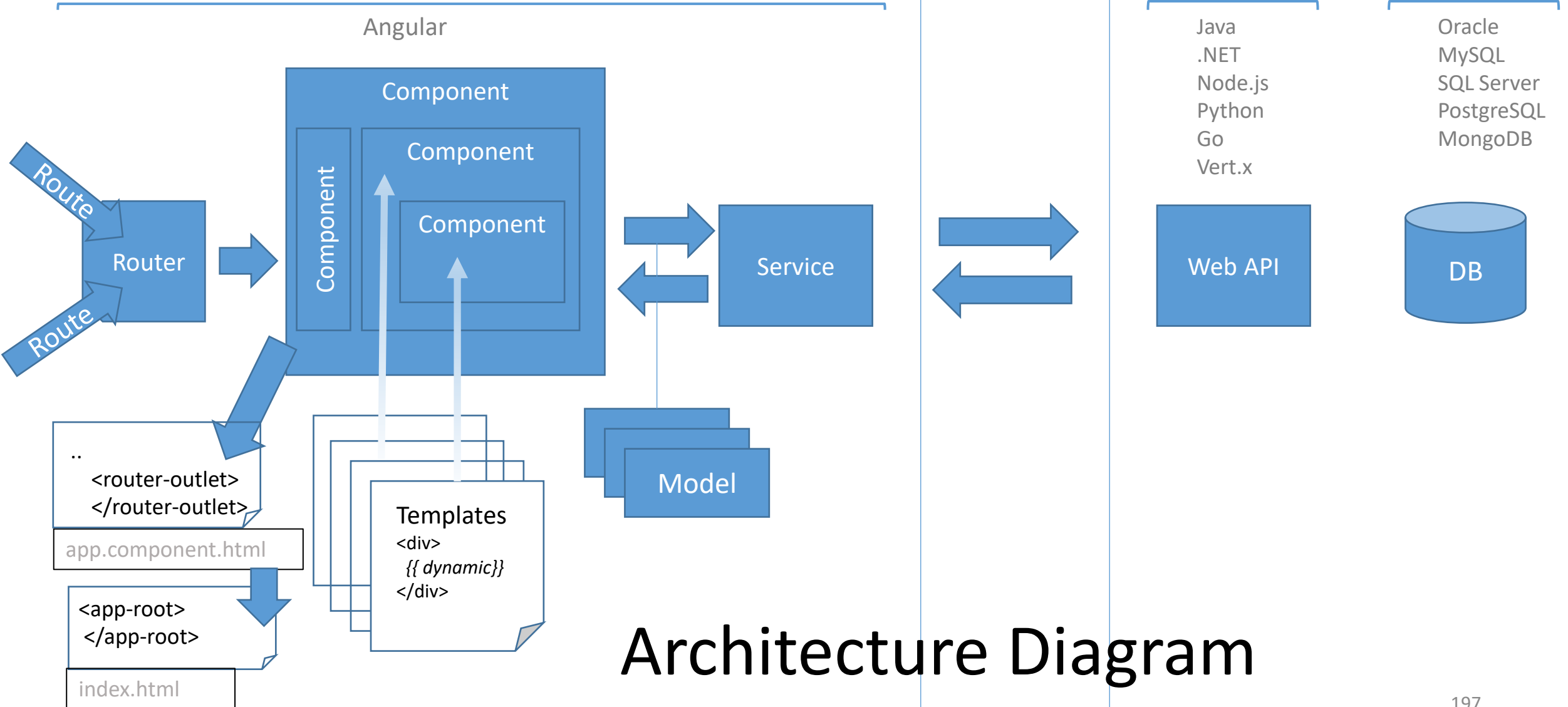Lab 25: Showing a Loading Indicator (optional)

# Routing & Navigation

Angular

Architecture Diagram

Router Overview

- A router watches the browser's url for changes and runs the corresponding code for that url (route).

- JavaScript applications commonly break the back button in the browser. A router can fix this problem.

- Before HTML5 there was no way to write to the browser's history via JavaScript

- The HTML5 history API (also know as pushState/replaceState) enables JavaScript code to add or modify history entries

- Before HTML5 history, JavaScript applications used the **fragment** identifier introduced by a hash mark # and is the optional last part of a **URL** for a document. It is typically used to identify a portion of that document (hyperlink bookmarks).

Client-side

Web Browser
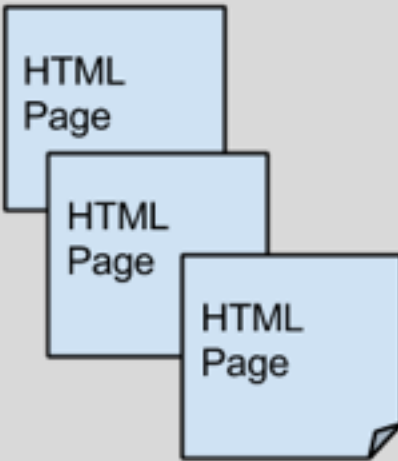
HTML Page

HTML Page

HTML Page

HTTP Request

HTML

Server-side

Web Server

HTML Templates + Data

Database

Server-side web application architecture

Component Router
Angular

- Navigates you through components and their corresponding client-side views without a page reload

- Bind the router to links on a page and it navigates to the appropriate application view when they are clicked

- Also can navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus

- Logs activity in the browser's history journal so the back and forward buttons work as expected

Single-page web application architecture

Routing Summary

- The application is provided with a configured router.

- The component has a RouterOutlet where it can display views produced by the router.

- It has RouterLinks that users can click to navigate via the router.

Router Configuration

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

import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home/home.component.ts'

const routes: Routes = [
  { path: 'home', component: HomeComponent }
];

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

Create an array of **Route** objects. Configure the **routes** in the app.

Create a **RouterModule** configured with the application routes.

The Angular CLI generates all this code except the Route objects.

We will learn more about modules next.

Router Outlet

```
<div class="container">
  <router-outlet>
  </router-outlet>
</div>
```
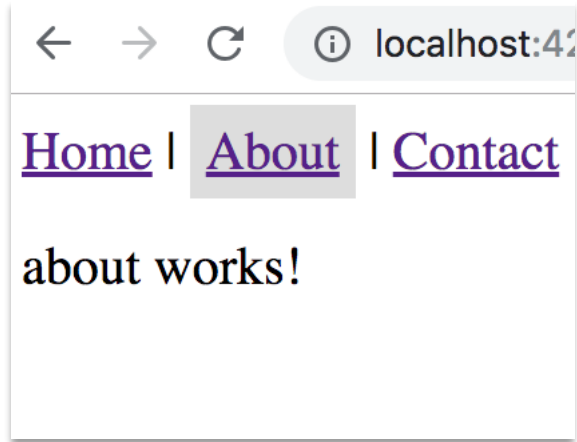
Placeholder where the component configured in the **Route** object gets rendered

Demo: Routing Basics

Instructor Only Demonstration

code\demos\routing-basics

Labs

Lab 26: Router Navigation
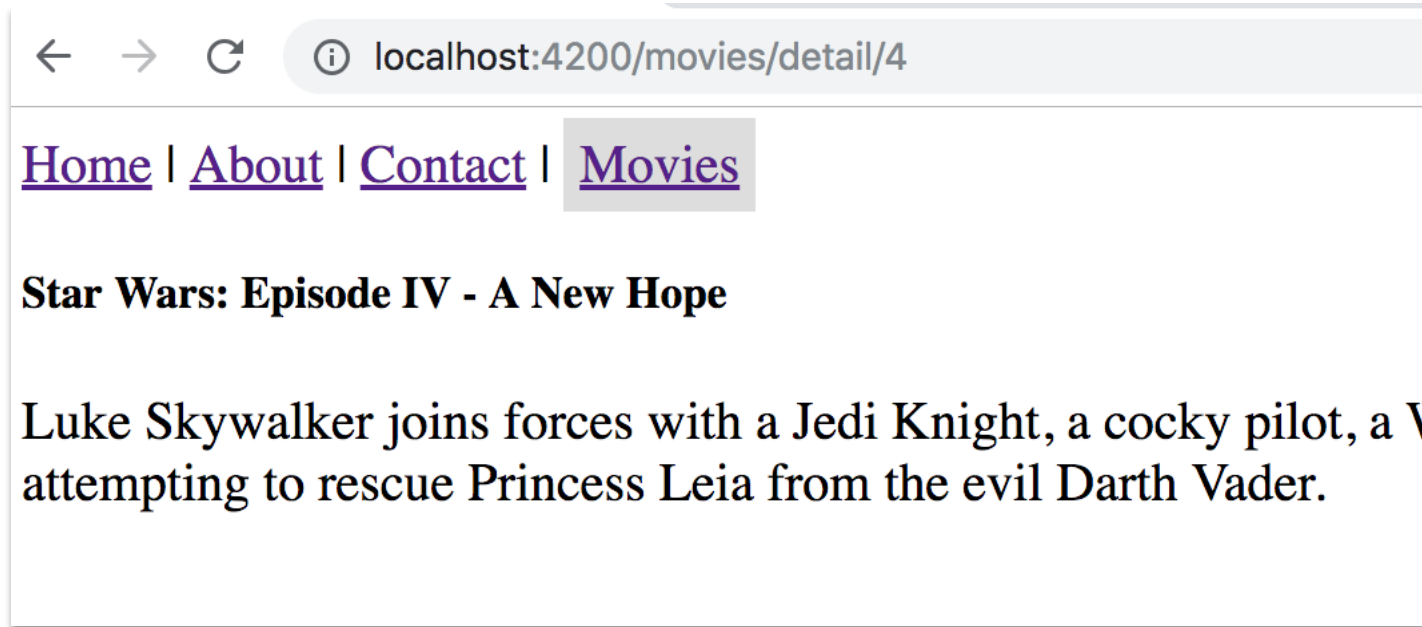
Navigating

- Using a routerLink directive
  - Route parameters
  - Query parameters
- Navigating with Code
  - Router
  - Route parameters
  - Query parameters
- Retrieving Parameters
  - ActivatedRoute
  - Synchronous
  - Asynchronous

Route Parameters or Query Parameters?

- There is no hard-and-fast rule. In general,
  - *prefer a route parameter when*
    - the value is required.
    - the value is necessary to distinguish one route path from another.
  - *prefer a query parameter when*
    - the value is optional.
    - the value is complex and/or multi-variate.

Matrix URL Notation

- The query string parameters are not separated by "?" and "&". They are **separated by semicolons (;)**

- This is *matrix URL* notation — something we may not have seen before.

- *Matrix URL* notation is an idea first floated in a [1996 proposal](#) by the founder of the web, Tim Berners-Lee.

Demo: Routing Navigation

Instructor Only Demonstration

code\demos\routing-navigation

Labs

Lab 27: Router Parameters

Build & Deploy

Using Angular CLI (Command-Line Interface)

Angular CLI: Builds

- Builds your application for deployment to production
  - Combining, minifying
  - Tree-shaking
  - Ahead-of-time compilation
  - Remove Decorators

CLI: Builds

- The **build** command sends output to the **dist** directory.
- **Copy** the contents of the **dist** directory to your web server to deploy.
- Run a development build

  ng build --dev

  Note: running just "ng build" defaults to a "dev" build

- Run a production build

  ng build --prod

- Passing the dev or prod flags effectively sets other flags
  - These flags are "meta" flags

Demo

Angular CLI: Builds

Instructor

Dev vs Prod Builds

Flag	--dev	--prod
--aot	false	true
--environment	dev	prod
--output-hashing	media	all
--sourcemaps	true	false
--extract-css	false	true
--named-chunks	true	false
--build-optimizer	false	true with AOT and Angular 5

Bundles

- runtime.bundle.js is webpack

- polyfills.bundle.js includes core-js and zone.js

- styles.bundle.js | .css includes all the CSS component styles as well as global styles in styles.css combined into one file

- vendor.bundle.js includes all the all the angular framework and rxjs
  - *when using build optimizer the vendor bundle is combined into the main bundle*

- main.bundle.js includes all your application code

```
chunk {0} runtime.ec2944dd8b20ec099bf3.js (runtime) 1.44 kB [entry] [rendered]
chunk {1} main.a51dad06404438ada278.js (main) 273 kB [initial] [rendered]
chunk {2} polyfills.f6ae3e8b63939c618130.js (polyfills) 59.6 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
```

Bundle Sizes

	dev (kB)	prod (kB)
runtime.bundle.js	5.83	1.45
main.bundle.js	27.90	239.00
polyfills.bundle.js	202.00	61.30
styles.bundle.js | css *	11.40	0.08
vendor.bundle.js	2,730.00	**
total	2,977.13	301.83

* Styles produce js file in dev mode but a css file in prod.

Ahead-of-Time (AOT) Compiler

- Angular offers two ways to compile your application:
  - *Just-in-Time* (JIT), which compiles your app in the browser at runtime
  - *Ahead-of-Time* (AOT), which compiles your app at build time.

- Converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase *before* the browser downloads and runs that code.

- The size of the Angular framework downloaded in the browser decreases in size by over 50%. Angular's template compiler code does not need to be sent to the browser because the template compilation has been done ahead-of-time.

Build Optimizer

- Build-Optimizer (PurifyPlugin) is a Webpack plugin created by the Angular team, specifically to optimize Webpack builds beyond what Webpack can do on its own.

- Optimizations
  - Removal of Angular decorators from AoT builds
  - adding /*@__PURE__*/ annotations to transpiled/downleveled TypeScript classes. The point of this is to make it easier for minfiers like Uglify to remove unused code.
  - Full list of optimizations is available here:
    - https://www.npmjs.com/package/@angular-devkit/build-optimizer

Build Optimizer and Vendor Chunk

- When using Build Optimizer the vendor chunk will be disabled by default. You can override this with --vendor-chunk=true.

- Total bundle sizes with Build Optimizer are smaller if there is no separate vendor chunk because having vendor code in the same chunk as app code makes it possible for Uglify to remove more unused code.

Labs

Lab 29: Build & Deploy

Appendices

The remainder of the topics in this manual are not always able to be covered during the introductory course. Instructors can choose to include them as time allows or questions arise. Students can use them as additional information to go deeper on topics particularly around the setup, build, and deployment of an Angular project.

Custom Pipe Example

```typescript
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({
  name: 'charlength'
})
export class CharacterLengthPipe implements PipeTransform{

  transform(value:string, length:number){
    return value.substring(0, length);
  }
}
```

Labs

Lab 28: Custom Pipe

Past, Present & Future

Features in Different Angular Versions

Angular 4

- Released 2017-03-23 (code named invisible-makeover)
- View Engine generates smaller code
- Enhanced *ngIf syntax
- Animation code now in own packages
- TypeScript 2.1
  - Improved compiler speed
  - Compliant with StrictNullChecks
- Universal
  - Run your Angular code on a server, not just browser
    - Faster initial render
    - Search Engine Optimization benefits for public sites
- Flat ESModules
  - More efficient tree shaking
- HttpClient (introduced in 4.3)

Angular 5

- Released 2017-11-01 (code named pentagonal-donut)
- Performance (Angular CLI v1.5)
  - Build optimizer
    - Webpack Plugin: require('@angular-devkit/build-optimizer').PurifyPlugin;
    - Additional optimizations (better tree shaking)
  - Angular Compiler
    - Faster builds and rebuilds, uses TypeScript transforms
  - Intl and Reflect polyfills no longer needed
  - Can configure (ts.config) preserveWhitespaces: false in template
- Features
  - upDateOn blur or submit
  - Router new lifecycle events
  - Package to faciliate adding service worker to apps (Progressive Web Apps)
- Upgrade tool
  - https://angular-update-guide.firebaseapp.com/

Angular 6

- Released May 3rd
- Aligning library releases (version numbers)
- Angular Elements
- Extending Angular CLI with Schematics
- RxJS 6 imports
- Service Providers
  - Instead of a module referencing services
  - Services reference modules

Aligning Library Releases

	Today	With v6
Angular	5.2.10	6.0
Material	5.2.4	6.0
CLI	1.7	6.0

Angular Elements

Wrap Angular Component as a
Custom Element (aka DOM Element)

Angular Elements: How it Works

You
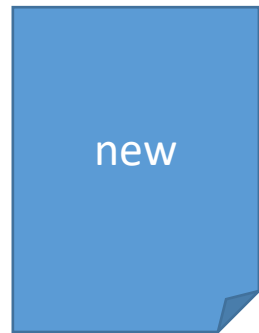
- Write components

- Wrap as Custom Element

They

- Import script

- Use component

Use Cases

- Use Angular component in another **Angular app**
- Use Angular components with **other libraries/frameworks**
  - Like a simple jQuery app or VueJS app
- Use Anguar components in **any app** written with **any technology**
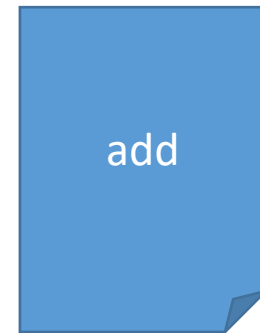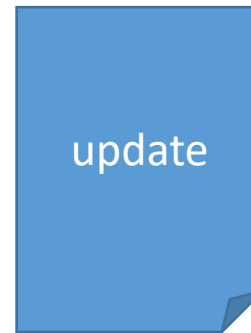
Extending Angular CLI with Schematics

ng [command]

new

generate

update

add

project

component
module
service
pipe
…

ng update

- Update
  - Npm dependencies
  - Code
  - V6: RxJS, Material

ng add

- Add capabilities
- Instant code and library scaffolding
- Supported by:
  - Angular Material (Ex. ng add @angular/material)
  - Angular Elements
  - Progressive Web App
  - Ng-bootstrap
  - Clarity
  - NativeScript
- Also, per-library 'generate' schematics

RxJS 5 Imports

Non-Exhaustive List

rxjs	rxjs/operators/audit	rxjs/operators/audit	There are a LOT
rxjs/Rx	rxjs/operators/auditTime	rxjs/operators/auditTime	
rxjs/Observable	rxjs/operators/buffer	rxjs/operators/buffer	
rxjs/Subject	rxjs/operators/bufferCount	rxjs/operators/bufferCount	
rxjs/BehaviorSubject	rxjs/operators/bufferTime	rxjs/operators/bufferTime	
rxjs/ReplaySubject	rxjs/operators/bufferToggle	rxjs/operators/bufferToggle	
rxjs/observable/interval	rxjs/operators/bufferWhen	rxjs/operators/bufferWhen	
rxjs/observable/timer	rxjs/operators/catch	rxjs/operators/catch	
rxjs/observable/from	rxjs/operators/combineAll	rxjs/operators/combineAll	
rxjs/observable/of	rxjs/operators/combineLatest	rxjs/operators/combineLatest	
rxjs/observable/fromEvent	rxjs/operators/concat	rxjs/operators/concat	
rxjs/observable/defer	rxjs/operators/concatAll	rxjs/operators/concatAll	
rxjs/observable/if	rxjs/operators/concatMap	rxjs/operators/concatMap	
rxjs/observable/throw	rxjs/operators/concatMapTo	rxjs/operators/concatMapTo	
rxjs/observable/empty	rxjs/operators/count	rxjs/operators/count	
rxjs/observable/never	rxjs/operators/debounce	rxjs/operators/debounce	
rxjs/observable/using	rxjs/operators/debounceTime	rxjs/operators/debounceTime	
rxjs/observable/generate	rxjs/operators/defaultIfEmpty	rxjs/operators/defaultIfEmpty	

RxJS 6 Imports

- rxjs: Contains creation methods, types, schedulers, and utilities

```
import {
  Observable,
  Subject,
  pipe,
  of,
  from,
  interval,
  merge,
  fromEvent,
} from 'rxjs';
```

- rxjs/operators: Contains all pipeable operators

```
import { map, filter, scan } from 'rxjs/operators';
```

Upgrading to Angular 7

- If already on Angular 6 and RxJS 6
  - ng update @angular/cli @angular/core

- If using Angular Material also run
 - ng update @angular/material

Angular 7

- Angular CLI prompts
- Angular Material CDK
  - ScrollingModule\ Virtual Scroll Viewport
  - DragDropModule
- Application Performance
  - Reflect-metadata polyfill not included in production builds
  - Angular CLI Budget Bundles
- Updated Dependencies
  - TypeScript 3.1
  - RxJS 6.3
  - Node 10 support
- Angular Elements now supports content projection using web standards (slots) for custom elements

Ivy

- Next generation Angular Renderer
- Backward compatible
  - No changes required for existing apps
- Benefits
  - Faster
    - Improve the speed of rendering
  - Smaller
    - Optimizes the size of the final package
  - Simpler
    - Human-readable code
    - Easy debugging (stack trace heaven)

Hello World
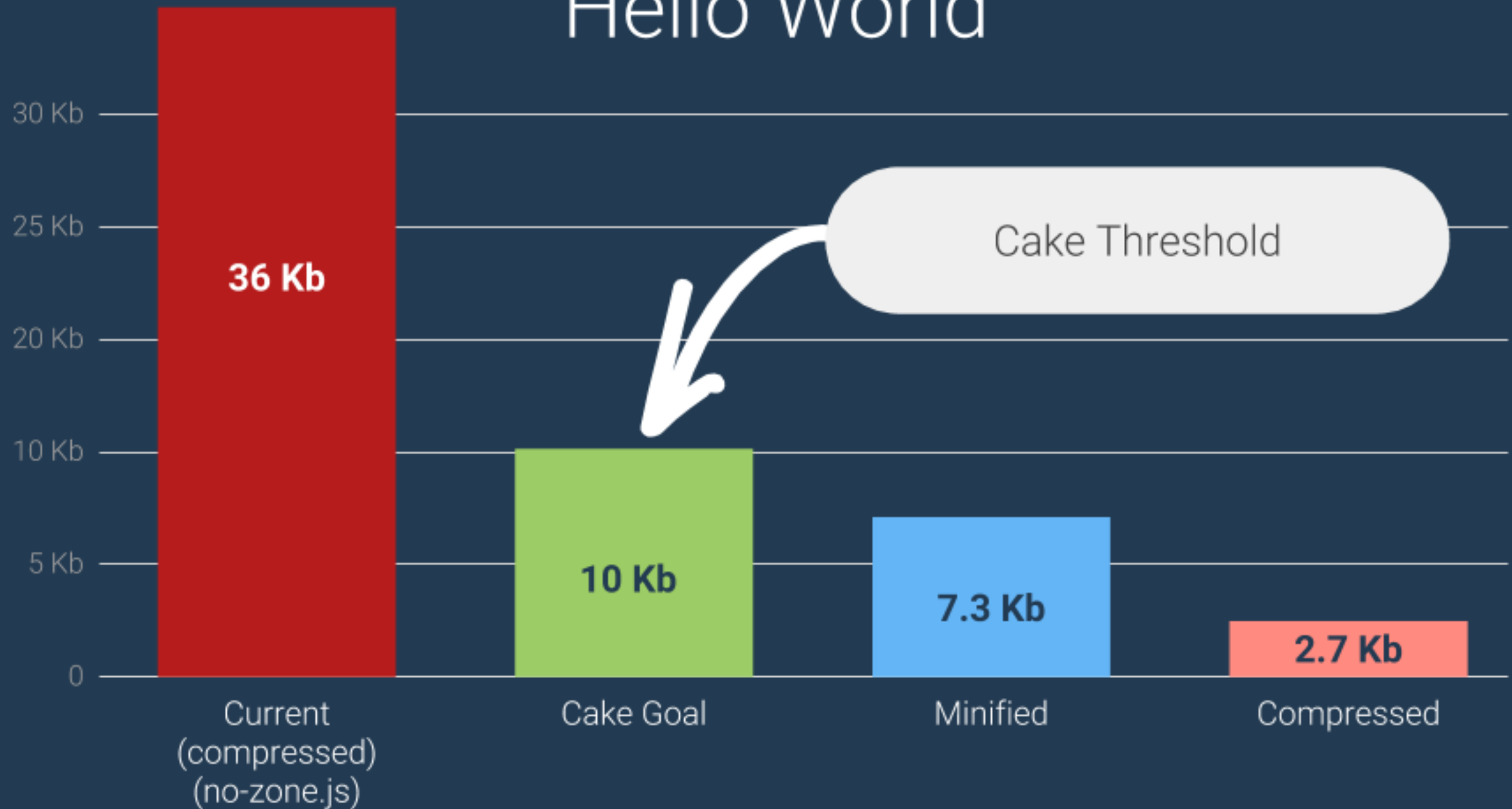
RxJS

RxJS

- **Reactive Extensions for JavaScript**

- RxJS is a library for composing asynchronous and event-based programs by using observable sequences.

- It provides one core type, the Observable, satellite types (Observer, Schedulers, Subjects) and operators inspired by Array#extras(map, filter, reduce, every, etc) to allow handling asynchronous events as collections

- Using RxJS, developers represent asynchronous data streams with Observables and query asynchronous data streams using the many operators (functions) provided

Observable

Time

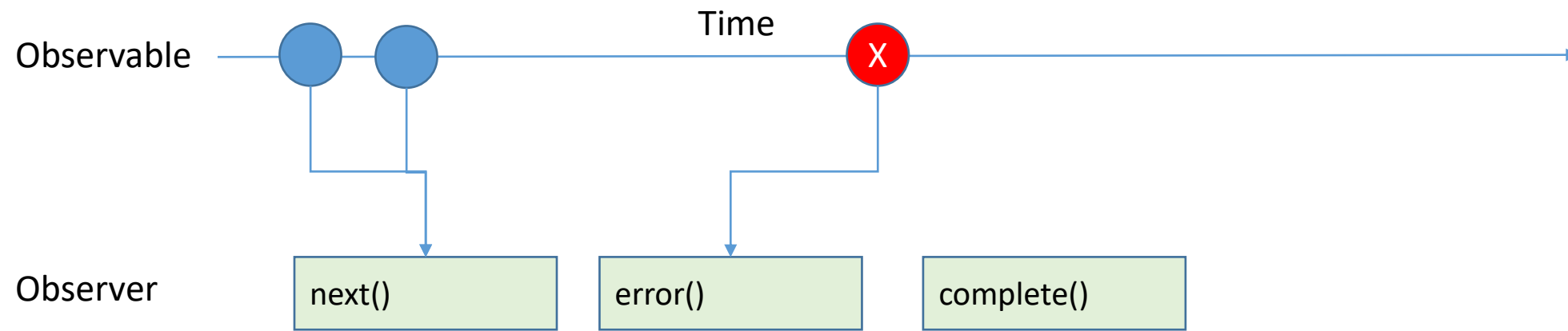Observer

next()

error()

complete()
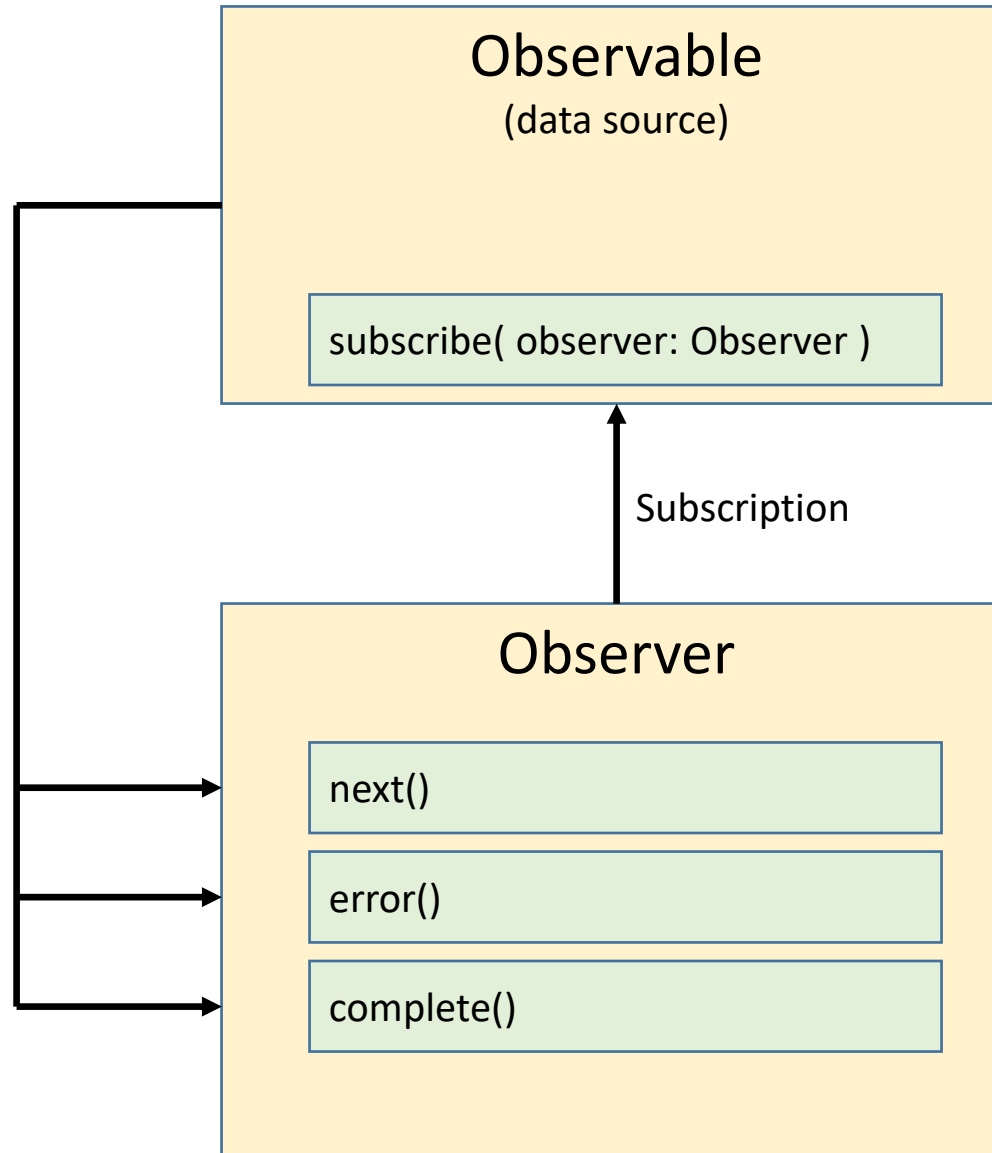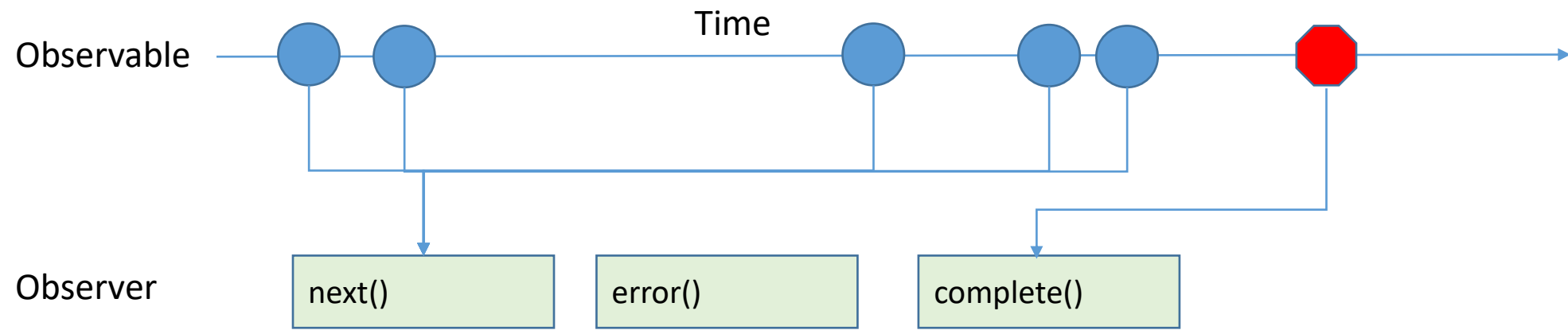
Observable

- A representation of any set of values over any amount of time

- The most basic building block of RxJS

- Represents a data source that streams values over time

- Observables are lazy push collections of multiple values

	Single	Multiple
Pull	Function	Iterator
Push	Promise	**Observable**

Observable Creation Functions

- Usually used to create Observables from scratch
- Pure functions attached to the Observable class in RxJS <=5.4
- Stand-alone functions in RxJS >=5.5 to RxJS<6
  - import { of } from 'rxjs/observable/of';
- RxJS >=6
  - import { of } from 'rxjs';

Creating Observables

```
of(1,2,3)
.subscribe(x=> console.log(x)); // 1, 2, 3


from([1,2,3])
.subscribe(x=> console.log(x)); // 1, 2, 3


create(internalObserver => {
  internalObserver.next('a');
  internalObserver.next('b');
})
.subscribe(x => console.log(x)); //a, b
```

Creating Observables

```
let button = document.querySelector("button");
fromEvent(button, "click")
.subscribe(x => console.log(x));
```

```
let input = document.querySelector("input");
fromEvent(input, "keyup")
.subscribe((x: Event) => console.log(x.target.value));
```

Demo: Obervables

code/demos/rxjs-observables

Observer

- A collection of callbacks that knows how to listen to values delivered by the Observable

- An Observer is a consumer of values delivered by an Observable

- Observers are simply a set of callbacks, one for each type of notification delivered by the Observable:
  - next
  - error
  - complete

Observer Example

```typescript
//Observer
let observer: Observer<any> = {
    next: x => console.log(x),
    complete: () => console.log('completed'),
    error: x => console.log(x)
}


of(1,2,3)

.subscribe(observer); // 1, 2, 3, completed
```

Demo: Observers

Instructor Only

code/demos/rxjs-observers

code/demos/rxjs-subscriptions
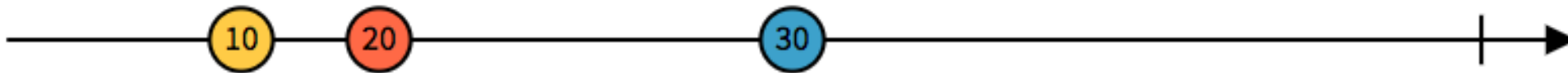
Operators

- Methods on Observable instances in RxJS <=5.4

- Stand-alone functions in RxJS >=5.5
  - import { map, filter } from 'rxjs/operators';
  - Use the this keyword to infer what is the input Observable

map Operator

```
//map returns same number of items as source
of(1, 2, 3)
  .pipe(map(x => x * 10))
  .subscribe(x => console.log(x));  //10, 20, 30
```


Source: rxmarbles.com

switchMap Operator

```
let obs1$ = of(1, 2, 3);
let obs2$ = of('a', 'b');

obs1$.pipe(switchMap(() => obs2$)).subscribe(observer);
// a, b, a, b, a, b, completed
```

```
let obs1$ = of(1, 2, 3);
let obs2$ = of('a', 'b');

obs1$.pipe(switchMap(() => obs2$, (n, l) => n + l)).subscribe(observer);
//1a, 1b, 2a, 2b, 3a, 3b
```

Demo: Operators

Instructor Only

code/demos/rxjs-operators

Subject

- A Subject is like an Observable, but can multicast to many Observers
- Subjects are like EventEmitters: they maintain a registry of many listeners
- Plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable)

Subject Example

```
var subject = new Subject();


subject.subscribe({
    next: (v) => console.log('observerA: ' + v)
});

subject.subscribe({
    next: (v) => console.log('observerB: ' + v)
});


subject.next(1);
subject.next(2);


// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

Practical Application of RxJS

```
this.items = this.searchTermStream.pipe(
  debounceTime(300),
  distinctUntilChanged(),
  switchMap((term: string) => this.wikipediaService.search(term))
);
```

- debounceTime waits for the user to stop typing for at least 300 milliseconds

- distinctUntilChanged ensures that the service is called only when the new search term is different from the previous search term

- switchMap calls the WikipediaService with a fresh, debounced search term and coordinates the stream(s) of service response

Demo: Practical Application of RxJS

Instructor Only

code/demos/rxjs-practical

EventEmitter or Observable

- Summary
  - Use EventEmitter in Components
  - Use some form of an Observable (Observable, Subject, BehaviorSubject) in Services

- Explanation
  - Do NOT count on EventEmitter continuing to be an Observable
  - Do NOT count on those Observable operators being there in the future
  - Only call event.emit()... Don't defeat angular's abstraction

Observables and Reactive Programming
In Angular

- We can structure our application to use Observables as the backbone of our data architecture

- Using Observables to structure our data is called Reactive Programming

- Reactive programming is programming with asynchronous data streams

- Observables are the main data structure we use to implement Reactive Programming.

Promises vs. Observables

Promises

- Returns single value

- Not cancellable

- Standard as of ES 2015

Observables

- Returns multiple values over time

- Cancellable

- Retry

- Supports map, filter, reduce and similar operators

- Proposed feature for ES 2016
  - Angular uses Reactive Extensions (RxJS)