Environment Setup

- Required Software
  - Git - https://git-scm.com/downloads
  - NodeJS - https://nodejs.org/en/download
  - Google Chrome - https://www.google.com/chrome
  - Visual Studio Code - https://code.visualstudio.com
- Online accounts
  - BitBucket - https://bitbucket.org
  - GitHub - https://github.com
  - GMail - https://gmail.com

Environment Setup

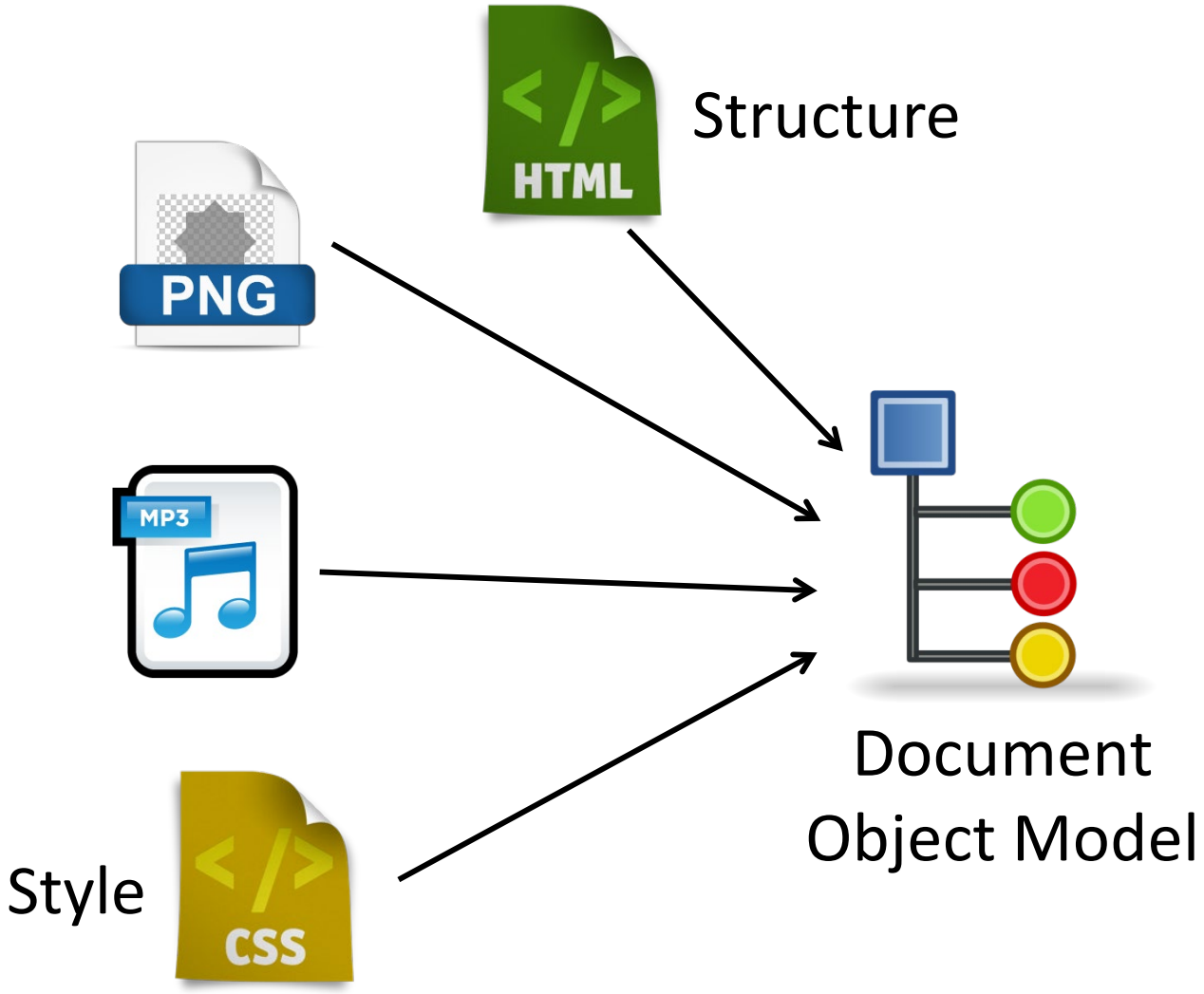
- Install TypeScript

```
npm install -g typescript
sudo npm install -g typescript
```

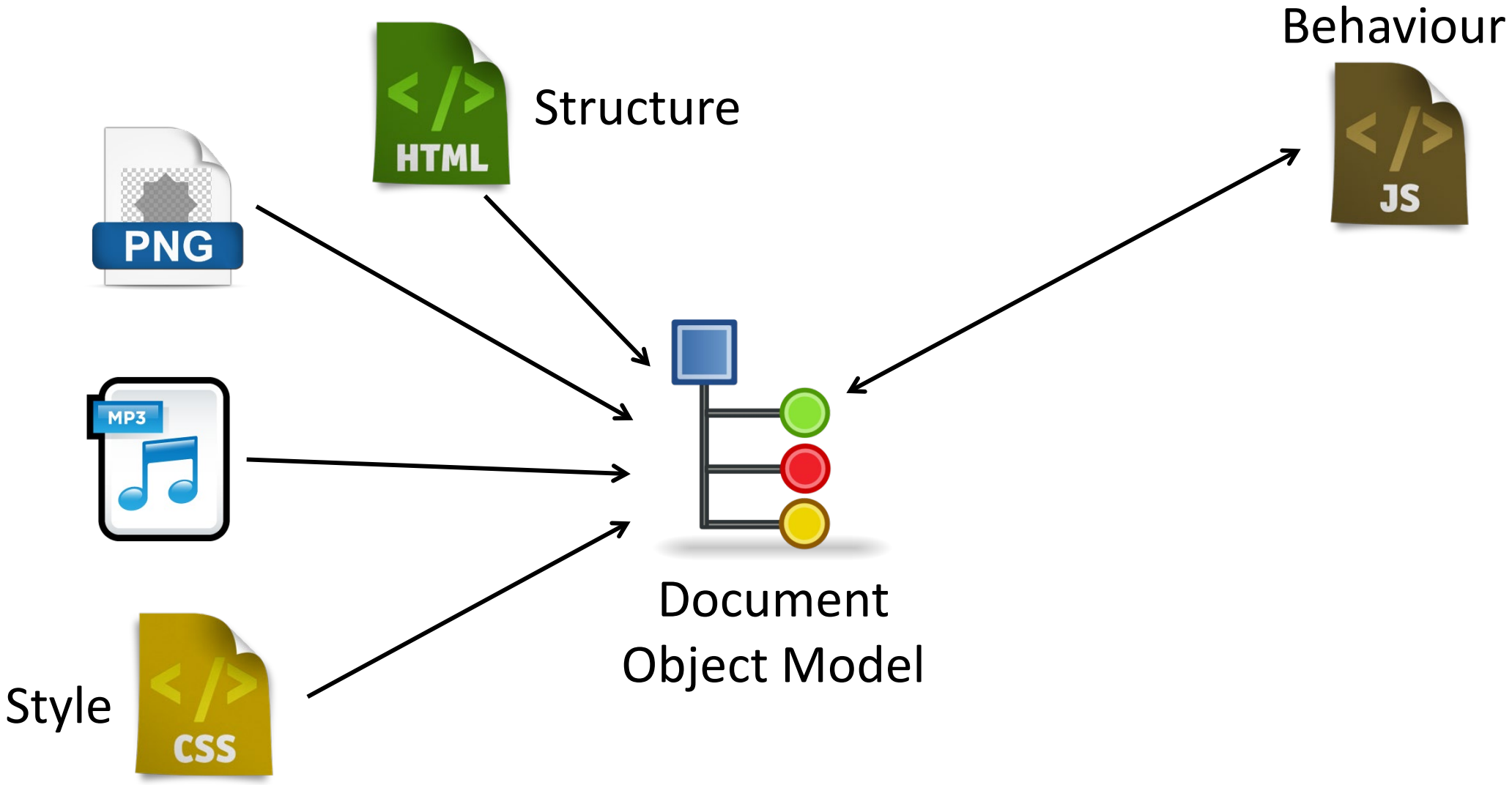
- Install Angular CLI
  - https://cli.angular.io

```
npm install -g @angular/cli
sudo npm install -g @angular/cli
```

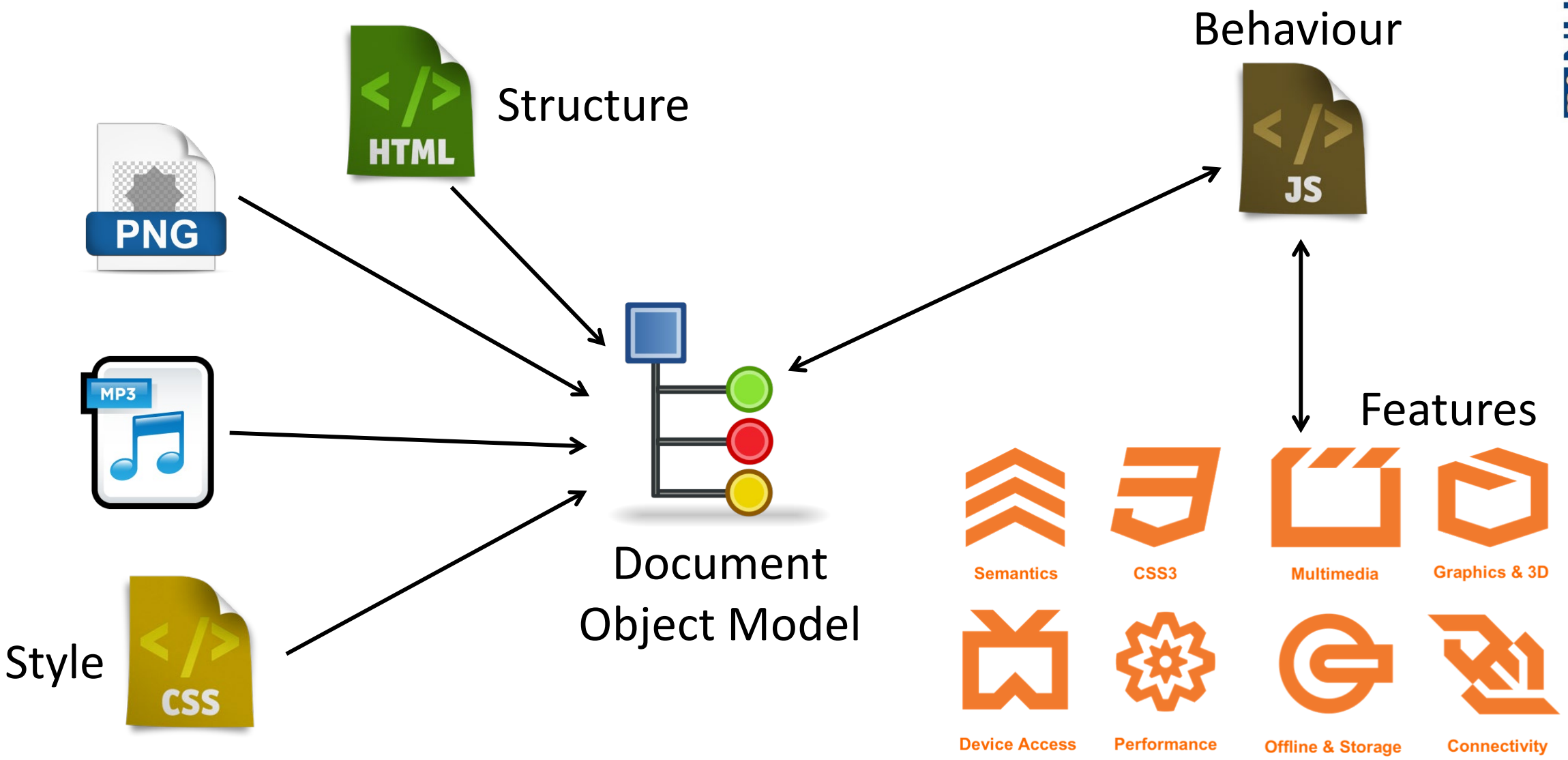
HTML5 Application



HTML5 Application

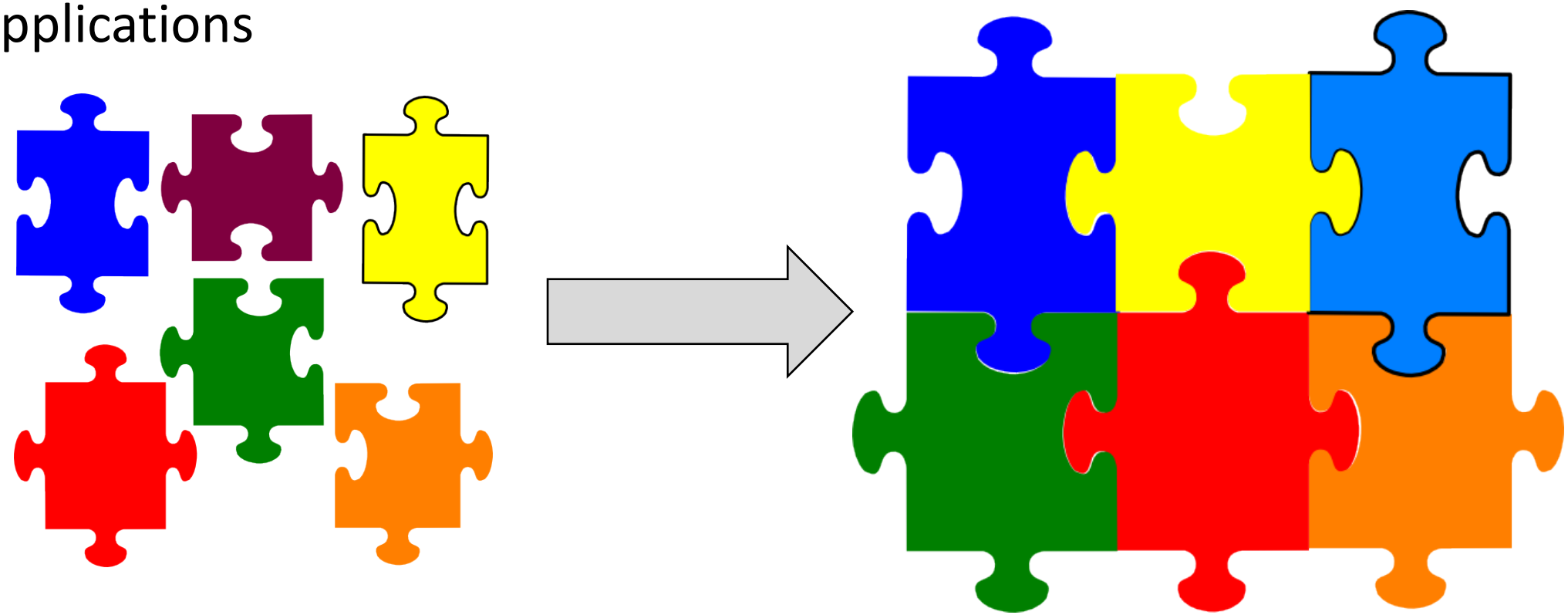


HTML5 Application



What is Angular?

- Is a component based TypeScript framework for developing HTML5 applications



Individual reusable components

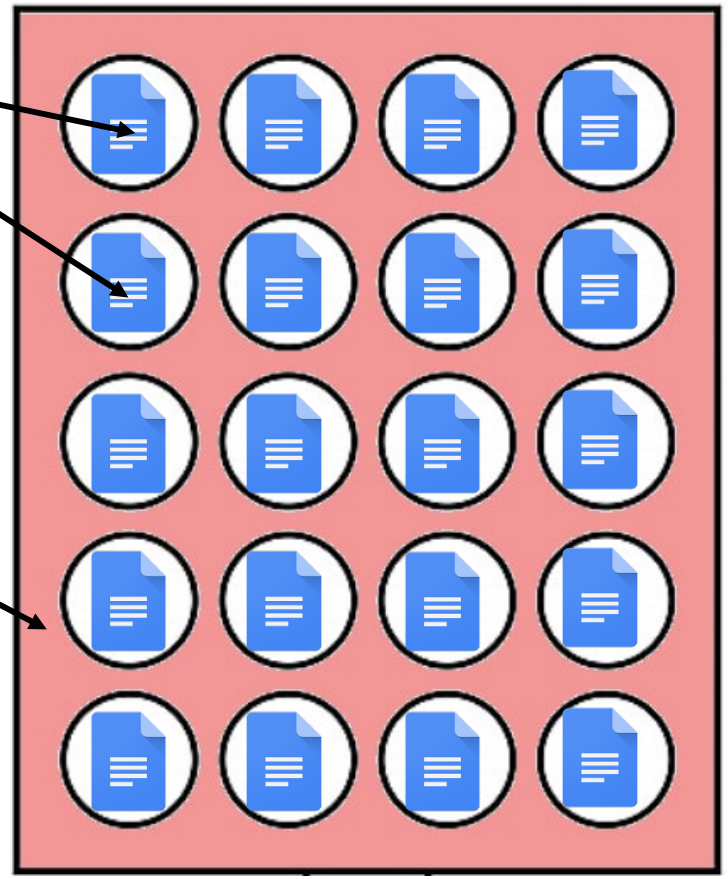
Assemble components into application

Image from https://www.clker.com/clipart-jigsaw-puzzle-6-pieces-5.html

Library vs Framework

You write this/these

The framework
use your code



Your program uses
the library

Library

Framework

Angular Framework

You write theses

Angular provided these

The Web/Browser

Angular Workflow

- Generate a new Angular application

```
ng new <app name>
```

- Add additional libraries

```
npm install --save <module>
```

- Start development server

```
ng serve -o
```

Angular Workflow

- Generate one or more components

```
ng generate component <name>
```

- Write one or more service
- Build final application

```
ng build --prod
```

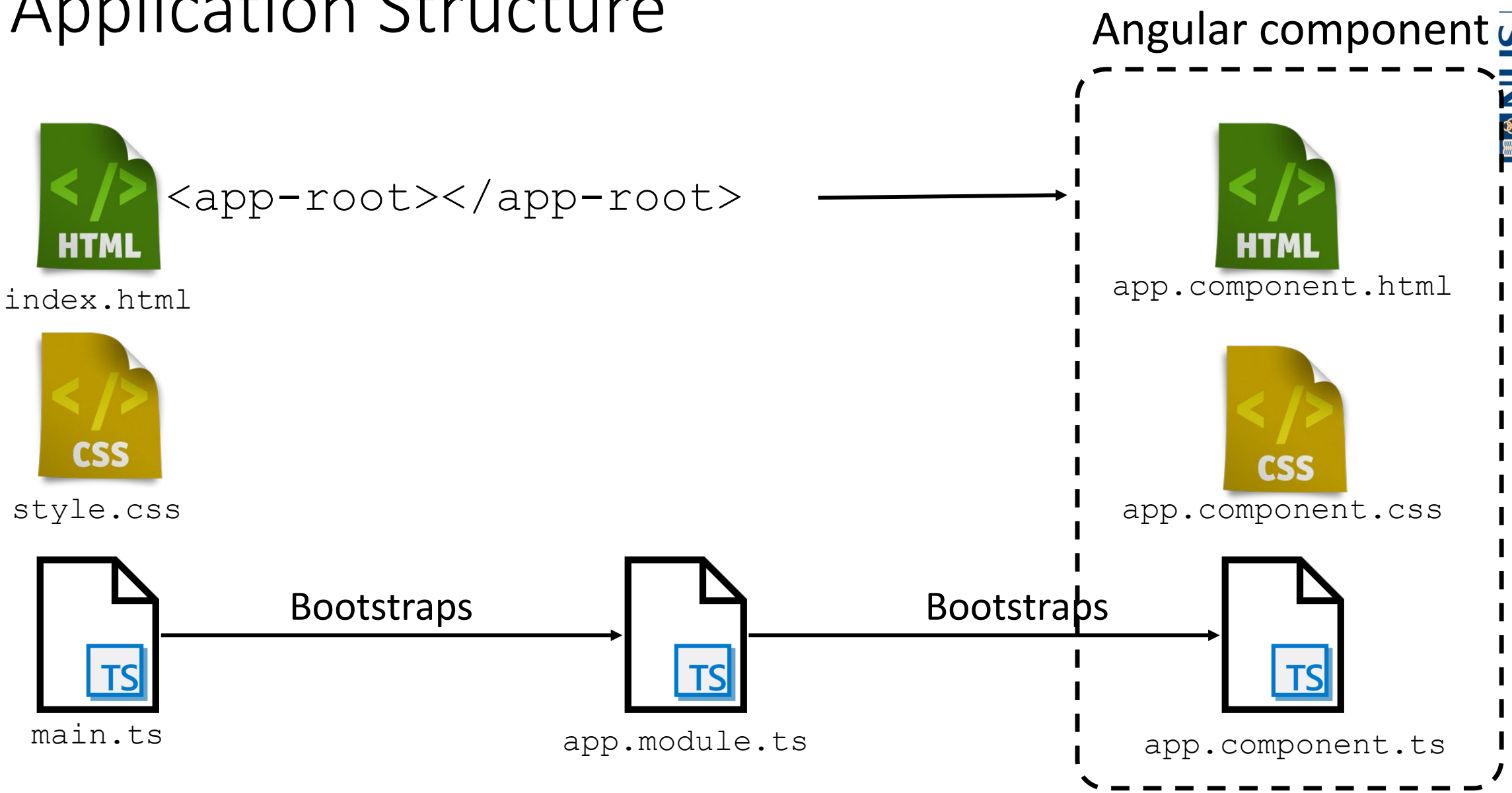
Angular Project Directory

- Generated by ng new
- Important files
  - `package.json` - list of installed modules in `node_modules`
  - `angular.json` - CLI configuration file
  - `src` - application source

`src` Directory

- Bootstrap
  - `index.html`, `main.ts`
- Global stylesheet
  - `styles.css`
- `asset` directory for images, etc.
- `app` directory
  - `app.module.ts`
  - `app.component.ts`, `app.component.css`, `app.component.html`

Application Structure



Application Structure

app.component.html

```
<h1>hello, world</h1>
```

app.component.css

```
h1 {
  color: red;
}
```

app.component.ts

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

index.html

```
<app-root></app-root>
```

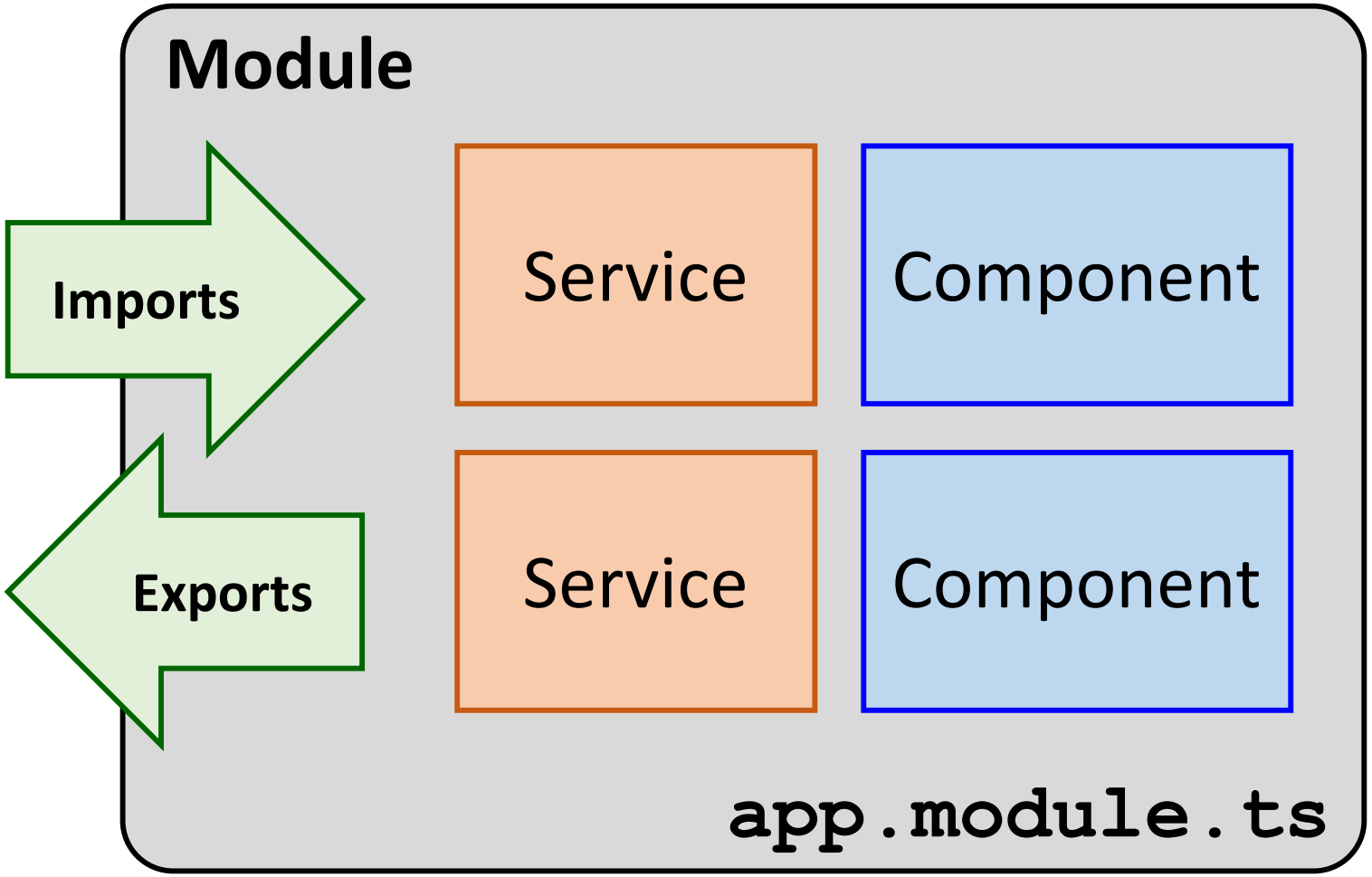
app.module.ts

```
@NgModule({
  ...
  bootstrap: [ AppComponent ]
})
```

main.ts

```
platformBrowserDynamic()
  .bootstrapModule(AppModule)
```

Module


Module
Imports
Exports
Service
Component
Service
Component
app.module.ts

Module

- A logical grouping of components, services, directives, pipes etc.
- Modules are opaque
  - The internals of a module is not visible to external unless a module chooses to export it
- Modules can use components from other modules by importing them
- `@NgModule` annotation is used to declare a class as module

Declaring a Module

Make a component available within the module

Make components, etc from this module available to other modules

The component to bootstrap if this module is bootstrapped/started

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

Make exported components from another module available in this module

Provide a service to all components and services within this module

Generating a Component

```
ng g c components/cart --spec false --flat
```

Updates AppModule

```
@NgModule({
  declarations: [CartComponent ],
})
export class AppModule {
```

app.module.ts

Generates these

```
src/app/components
  cart.component.ts
  cart.component.html
  cart.component.css
```

Component class

cart.component.ts

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

Component

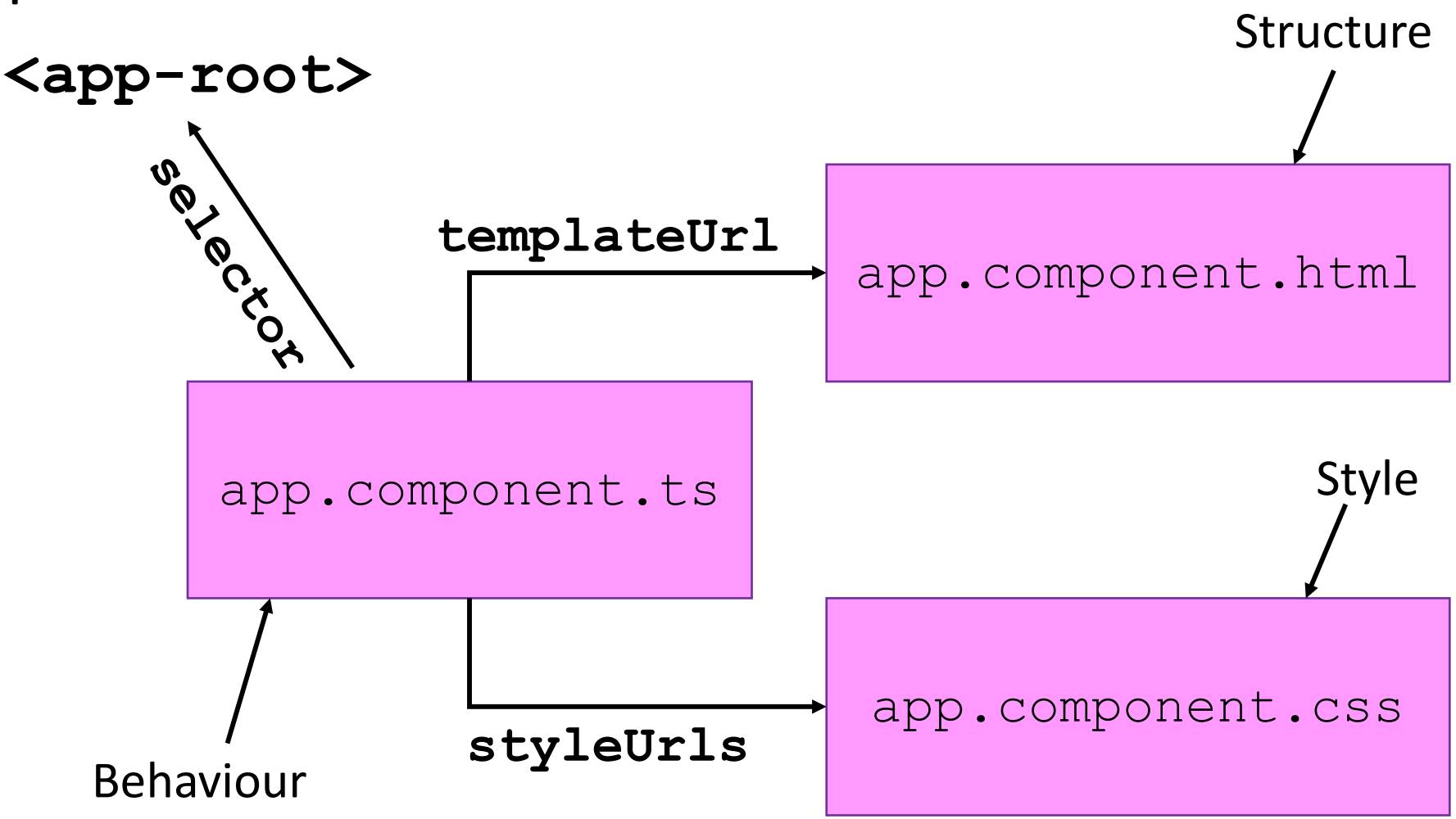
The 'tag' that instantiates this component

The HTML that defines the component's structure

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

One or more CSS that defines the component's style

Component


<app-root>
selector
templateUrl
app.component.html
Structure
app.component.ts
Style
app.component.css
styleUrls
Behaviour

Accessing Properties

- Properties/members in the component class can be accessible by its template
- Properties are displayed by `{{ }}`
- Changes in the properties are immediately reflected in the template

```
@Component({
  templateUrl:
    'app.component.html'
    ...
})
export class AppComponent {
  title = 'hello, world';
}
```

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

Angular expression

Property Binding

- Any HTML attribute can be bound to the component's properties by surrounding the attribute name with [ ]

```
<input [value]="name">

<button type="button" [disabled]="isDisabled">
</button>

<p [style.font-size]="fontSize">
  {{ greetings }}
</p>
```

Event Binding

- HTML element generate events
  - Clicked, mouse hover, value changed, etc
  - See https://developer.mozilla.org/en-US/docs/Web/Events
- Bind HTML event with `( )` to a method/function in the component's class
  - Drop the on when binding to events
  - eg `onClick` becomes `(click)`
- Pass `$event` into the function to get the event object

Event Binding

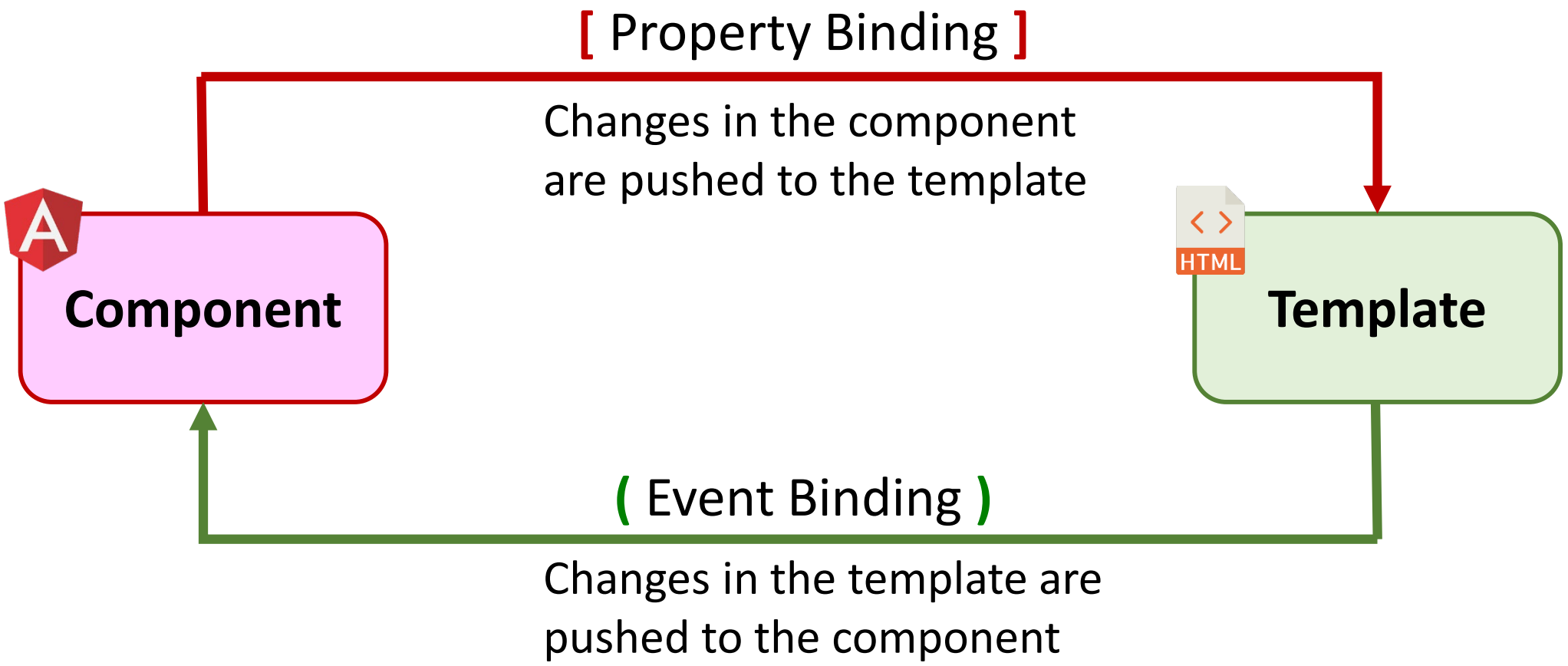
```
<p [style.font-size]="fontSize">
  {{ greetings }}
</p>
<input type="range"
  min="1" max="10" step="0.1"
  (change)="fontSizeChanged($event)">
```

This is DOM event object. See https://developer.mozilla.org/en-US/docs/Web/API/Event

Method is called whenever the value of the slider changes

```
export class AppComponent {
  greetings = 'hello world';
  fontSize = '1em';
  fontSizeChanged($event) {
    this.fontSize = `${$event.target.value}em`;
  }
}
```

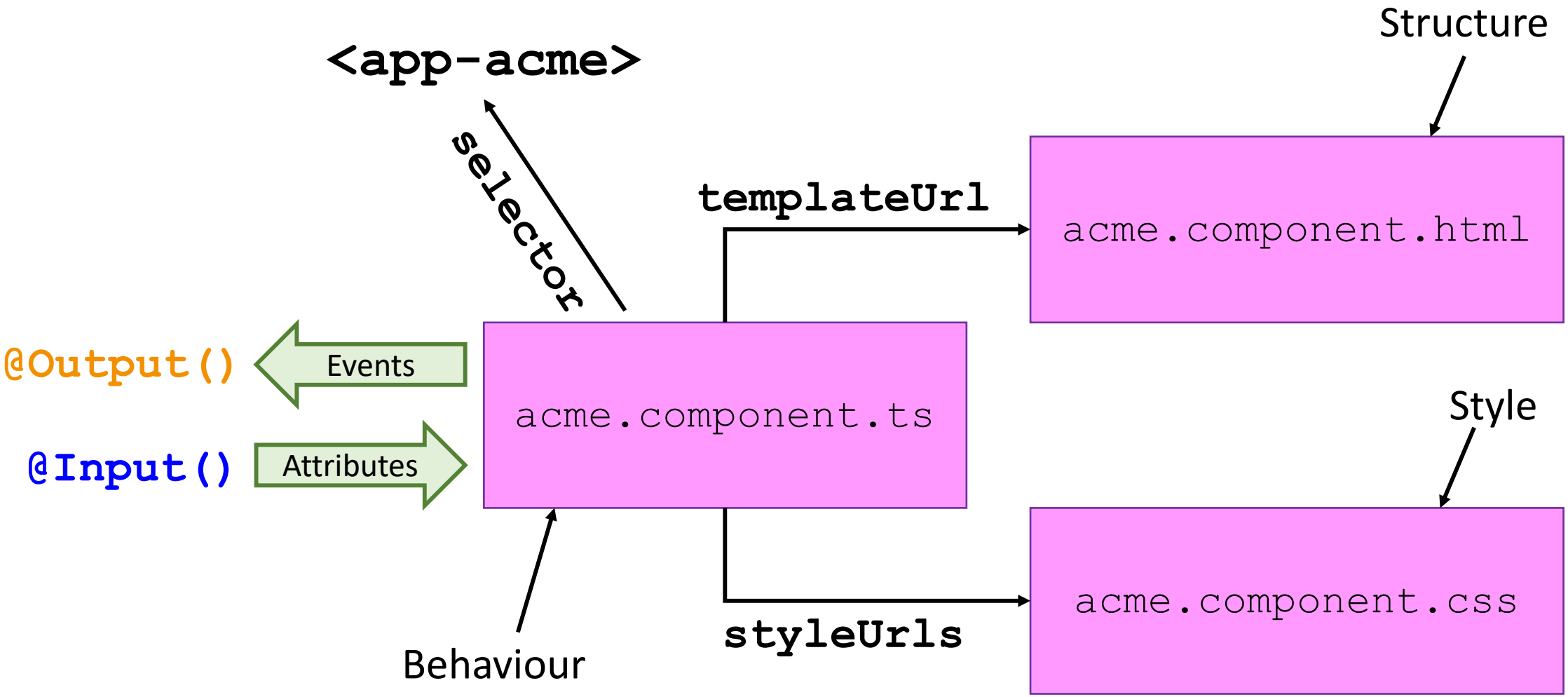
Property and Event Binding



Component

- Components internals are not accessible from the outside of the component
  - Eg accessed by other components
- Declare properties and events
  - To allow binding by other components
- Annotate class member with
  - `@Input()` for attribute
  - `@Output()` for event - `EventEmitter` type

Component


<app-acme>
selector
templateUrl
acme.component.html
Structure
@Output()
Events
@Input()
Attributes
acme.component.ts
Style
acme.component.css
styleUrls
Behaviour

Example - font-size.component.html

```
<div>
  <h2 [style.font-size]="fontSize">
    {{ message }}
  </h2>
</div>

<div>
  Font size:
  <input type="range" min="1" max="10" step="0.1"
      (change)="fontSizeChanged($event)">
</div>
```

Example - font-size.component.ts

```
@Component({
   selector: 'app-font-size',
   templateUrl: './font-size.component.html',
   styleUrls: [ './font-size.component.css' ]
})
export class FontSizeComponent {
   @Input() message: string;
   @Output() onFontSize = new EventEmitter<number>();
   fontSize: string = '1em';

   fontSizeChanged($event) {
      const fontSize = parseInt($event.target.value);
      this.fontSize = `${fontSize}em`;
      this.onFontSize.next(fontSize);
   }
}
```

Define an externally accessible attribute call `message`

Define an event called `onFontSize`

The event object viz. the value that this event is firing

Fire the event with the latest font size

Example

font-size.component

```
<h2>{{ message }}</h2>
<input (change)="fontSizeChanged($event)">
```

```
export class FontSizeComponent {
  @Input() message;
  @Output() onFontSize = new EventEmitter<number>();
  fontSizeChanged($event) {
    this.onFontSize.next($parseInt(event.target.value));
  }
```

app.component

```
<app-font-size
  [message]="title" (onFontSize)="sizeChanged($event)">
</app-font-size>
```

```
export class AppComponent {
  title = 'hello, world';
  sizeChanged(size) {
    console.log(`font size: ${size}`);
  }
```

@Output

- `@Output` annotation to declare an event to be fired by the component
- Type is `EventEmitter` and also event object

```
@Output() onEvent = new EventEmitter<string>();
```

- To fire an event

```
this.onEvent.next('hello');
```

Directives

- Angular's way of extending HTML capabilities
  - Eg. conditionally apply CSS to HTML element
- Directives typically starts with `ng`
  - Eg. `ngFor`, `ngIf`, `ngClass`, etc.
- Two types of directives
  - Non structural - enhances an element
  - Structural - add or remves HTML element; prefixed with a `*`
    - eg `*ngIf`

ngClass - Conditional Styling

```
<input type="text"
   [disabled]="isDisabled"
   [ngClass]=
    "{ 'grey-border': isDisabled, 'blue-border': !isDisabled }">
```

Conditionally added these classes based on the boolean variable

*ngIf - Conditional Display

```
<div *ngIf="cart.length <= 0">
 Your cart is empty
</div>

<div *ngIf="cart.length > 0">
 Your cart has {{ cart.length }} item(s)
</div>
```

Conditionally add or remove these elements

*ngFor - Loops

Array

Optional index

Generate the element that is annotated with *ngFor

```
<ul>
  <li
     *ngFor="let item of cart; let i = index;"
     [ngClass]="{ shaded: (i % 2) }">
    {{ item }}
  </li>
</ul>
```

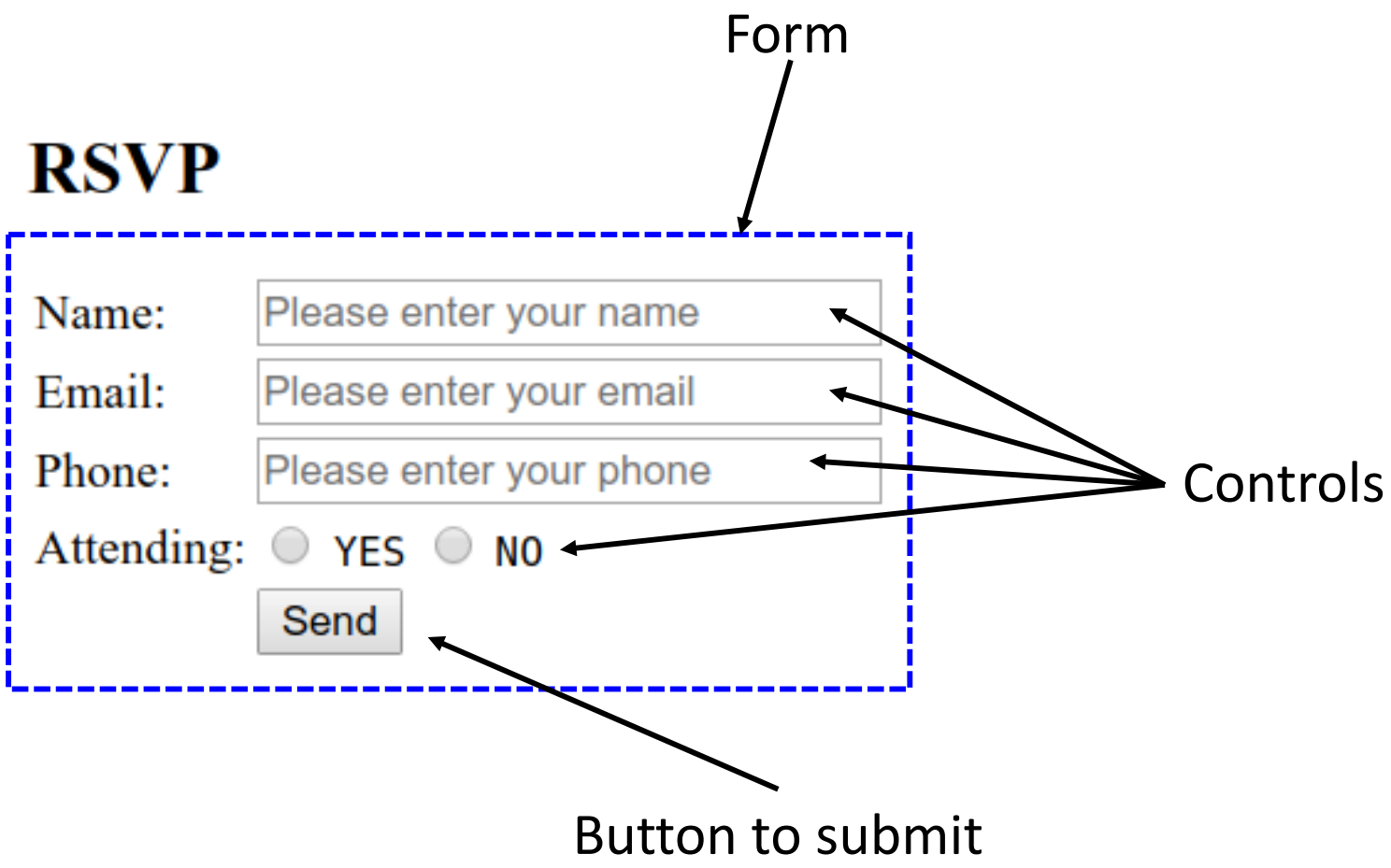
Generate HTML element from the contents of an array

Configuring Forms Module

- `FormsModule` are optional. Need to be added the module

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ]
})
export class AppModule {
  ...
```

Form


Form
RSVP
Name:
Please enter your name
Email:
Please enter your email
Phone:
Please enter your phone
Attending:
YES
NO
Send
Controls
Button to submit

Form

Form

```
<form>

  Name: <input type="text" name="name">

  ...

  Attending:
  <input type="radio" name="attending" value="yes"> YES
  <input type="radio" name="attending" value="no"> NO

  <button type="submit">
    Send
  </button>

</form>
```

Controls

Button to submit

Angular Forms

- Angular creates an `NgForm` component for every `<form>`
  - Fires an event call `ngSubmit` in response to the submit
  - Form component is called `ngForm`

Event fired by `NgForm` component when button (submit) is pressed

ngForm

```
<form (ngSubmit)="processForm()">

  <button type="submit">
    Send
  </button>

</form>
```

A `NgForm` is instantiated and mapped to a form

Forms - NgModel

- Annotate form fields `<input>` with `ngModel` directive
- Allow `NgForm` to manage them as controls
- Every field must have a unique name

```
<input type="text" name="email" ngModel>
```

Form

```
<form (ngSubmit)="processForm()">

  Name: <input type="text" name="name" ngModel>

  ...

  Attending:
  <input type="radio" name="attending" value="yes" ngModel> YES
  <input type="radio" name="attending" value="no" ngModel> NO

  <button type="submit">
    Send
  </button>

</form>
```

Template Reference

- A variable that references a HTML element

```
<h1 #h1Element>hello, world</h1>

<form #form>...</form>
```

- Assign the template reference on a `<form>` to `ngForm` object
  - Access to the form when it is submitted

```
<form #form="ngForm" (ngSubmit)="processForm(form)">
  ...
</form>
```

Processing Form

```
<form #form="ngForm" (ngSubmit)="processForm(form)">

 Name: <input type="text" name="name" ngModel>


   export class RSVPComponent {

     processForm(form: NgForm) {
       const name = form.value.name;
       ...
       form.reset();
     }
   }
```

Pass the form into the event handler

Clears the values in the form

Form Validation

- Form controls can be marked mandatory by adding the required attribute
  - Tracked by Angular, must also have `ngModel` directive
- Properties from the `NgForm` template reference provides validity status eg `form.invalid`
  - `valid`, `invalid`, `pristine`, `dirty`, `touched`, `untouched`
- The following directives can be used to perform further checks on the form controls
  - `email`, `minlength`, `maxlength`, `pattern`, `max`, `min`

Form Validation

```
<form #form="ngForm" (ngSubmit)="register(form)">
  Email:
  <input type="email" name="email" required
      email ngModel>

  Phone:
  <input type="tel" name="phone" required
      minlength="8" ngModel>

  <button type="submit" [disabled]="form.invalid">
    Register Me
  </button>
</form>`
```

Annotate the controls with required to indicated required values. Must have `ngModel` directive

Additional constraints

Use property binding to disable the form when the form is invalid

Component Lifecycle

- Angular creates, renders and destroy the component
- Lifecycle hooks allow you to perform certain operations at these key moments
  - Eg. load data before the component is destroyed
- Implement one or more of these lifecycle interfaces
  - `OnInit` - called just after the component is created but before displaying
  - `OnDestroy` - called just before the component is destroyed

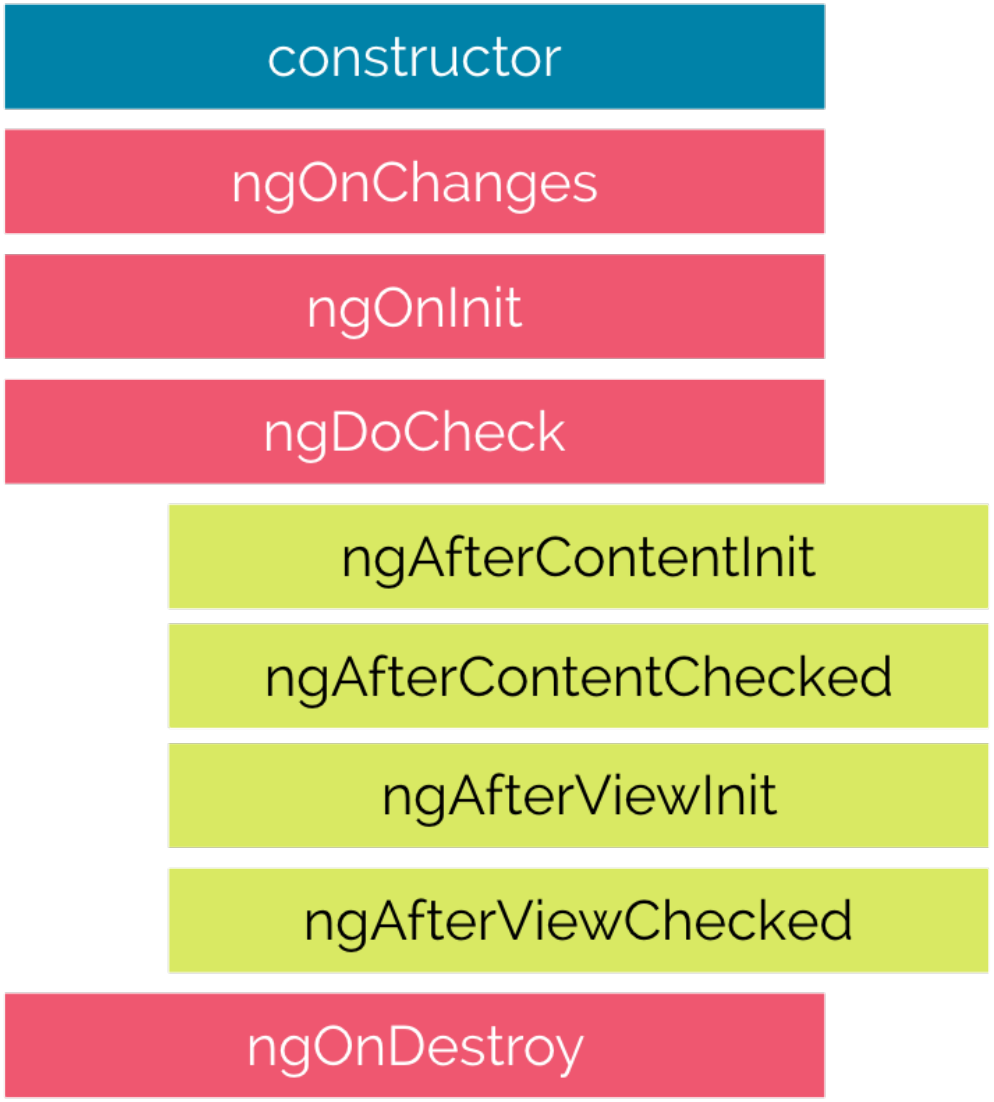


Image from https://codecraft.tv/courses/angular/components/lifecycle-hooks/

Using Lifecycle Hooks

```
import { OnInit, OnDestroy } from '@angular/core';

export class AppComponent implements OnInit, OnDestroy {

   ngOnInit() {
      //From OnInit interface

   }

   ngOnDestroy() {
      //From OnDestroy interface

   }
}
```

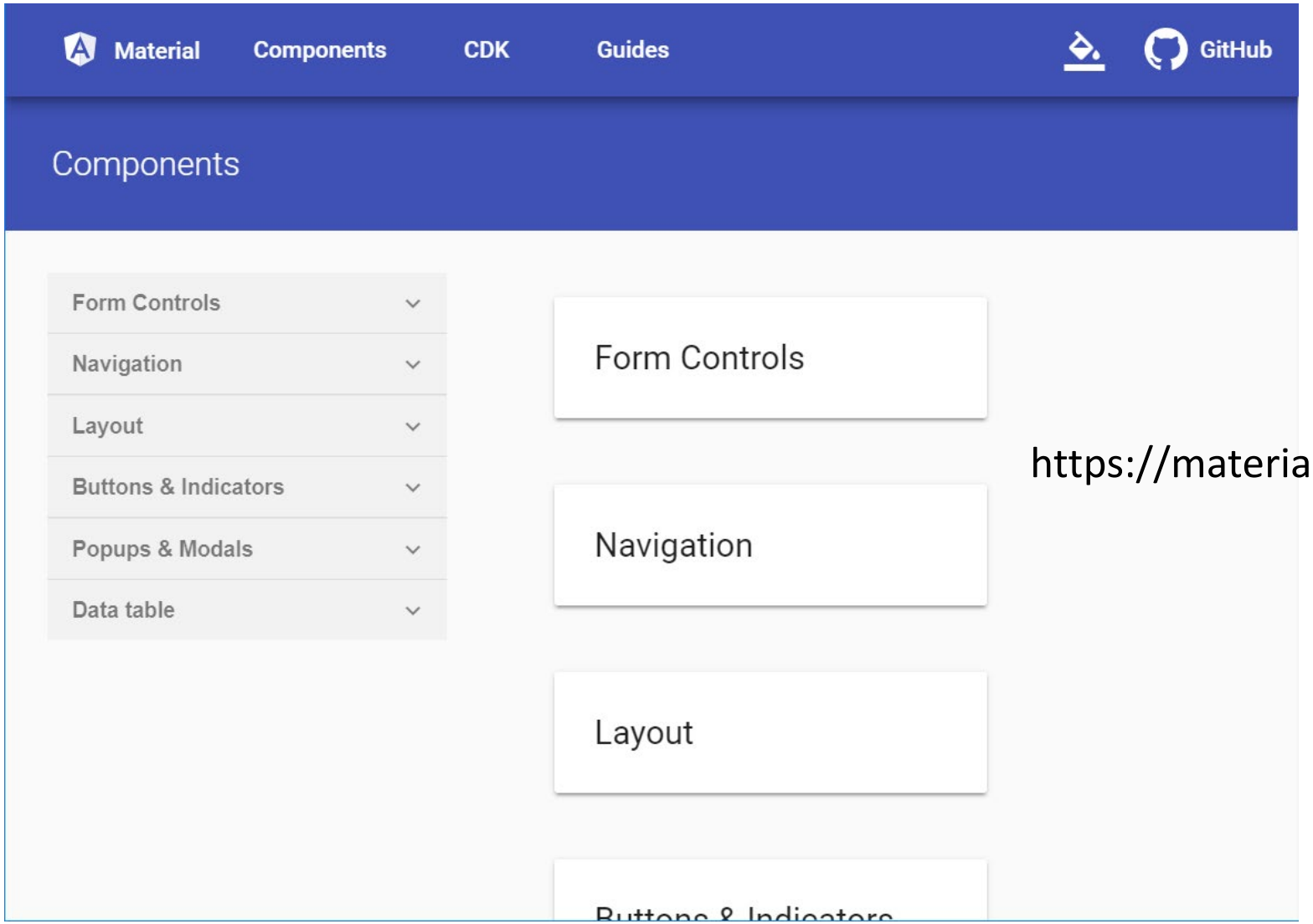
`ngOnInit` will be called just after component is created

`ngOnDestroy` will be called just before component is destroyed

Angular Material

- Set of UI components for Google's Material Design Specification
- Consist of Component Development Kit and Material components
  - Dependent on animations and hammer.js
- Angular Materials Setup
  - See https://material.angular.io/guide/getting-started for details
- Steps
  - Install `@angular/cdk`, `@angular/material`, `@angular/animations`, `hammerjs`
  - Import `BrowserAnimationsModule` in `app.module.ts`
  - Select a theme from `@angular/material/prebuilt-themes`
  - Import `hammer.js` in `main.ts` for gesture support
  - Add Material icons

Angular Materials Documentation Page



https://material.angular.io/components/categories

Using Material Components

- Material component consist of one or more material markup
  - Eg <button mat-button>, <mat-icon>
- Almost every component is in a separate module
  - Need to import module to unlock component
- Example: `mat-button`, `mat-raised-button`, `mat-fab`
  - From `MatButtonModule`

Using Angular Material Components

```
import { MatButtonModule } from '@angular/material/button';
import { MatIconModule } from '@angular/material/icon';
const MATERIAL = [ MatButtonModule, MatIconModule ];
@NgModule({
   imports: MATERIAL,
   exports: MATERIAL
})
export class MaterialModule { }
```

Keep Material module imports in a separate module

```
import { MaterialModule } from './material.module';
@NgModule({
   imports: [ BrowserModule, MaterialModule ],
   ...
})
export class AppModule {
```

Import `MaterialModule` into the `AppModule`

Example - Buttons

Press

```
<button type="button"
     mat-button>
   Press
</button>
```

Press

```
<button type="button"
     mat-raised-button>
   Press
</button>
```

Press

```
<button type="button"
     color="primary"
     mat-raised-button>
   Press
</button>
```

```
<button type="button"
     mat-icon-button>
   <mat-icon>favorite<mat-icon>
</button>
```

```
<button type="button"
     color="accent"
     mat-raised-button
     mat-icon-button>
   <mat-icon>favorite<mat-icon>
</button>
```

Coffee

```
<button type="button"
     color="warn"
     mat-button>
   <mat-icon>
     free_breakfast
   <mat-icon>
</button>
```

Form Field

`mat-form-field` are used to wrap `input` and `textarea`

```
<form #form
   (ngForm)="process(form)">

  <mat-form-field>
    <input type="email" name="email"
         placeholder="Please enter your email"
         ngModel matInput>
  </mat-form-field>

</form>
```

Please enter your email

Blur

Please enter your email

Focus

Radio Button


```
<mat-radio-group name="attending" ngModel>
  <mat-radio-button value="yes">
    Yes
  </mat-radio-button>
  <mat-radio-button value="no">
    No
  </mat-radio-button>
</mat-radio-group>
```

Select

Guest

```
<mat-form-field>
  <mat-select placeholder="Guest" name="guest" ngModel>
    <mat-option *ngFor="let g of [0, 1, 2, 3]"
        [value]="g">
      {{ g }}
    </mat-option>
  </mat-select>
</mat-form-field>
```

Checkbox

☐ Yes, I wish to receive newsletter

```
<mat-checkbox name="newsletter" ngModel>
  Yes, I wish to receive newsletter
</mat-checkbox>
```

Checkbox - Multiple Values

Diet:

☐ Fish ☐ Meat ☐ Vegetables

MatCheckboxChange
is the event object

```
<mat-checkbox ngModel (change)="diet[0] = $event.checked">
  Fish
</mat-checkbox>
<mat-checkbox ngModel (change)="diet[1] = $event.checked">
  Meat
</mat-checkbox>
<mat-checkbox ngModel (ngModel)="diet[2] = $event.checked">
  Vegetables
</mat-checkbox>
```

```
diet = [ false, false, false ]
```

Array elements will be set to true when checkbox is selected

Date Picker

- Datepicker component uses moment for date and time
  - See https://momentjs.com
- Will have to install moment adapter

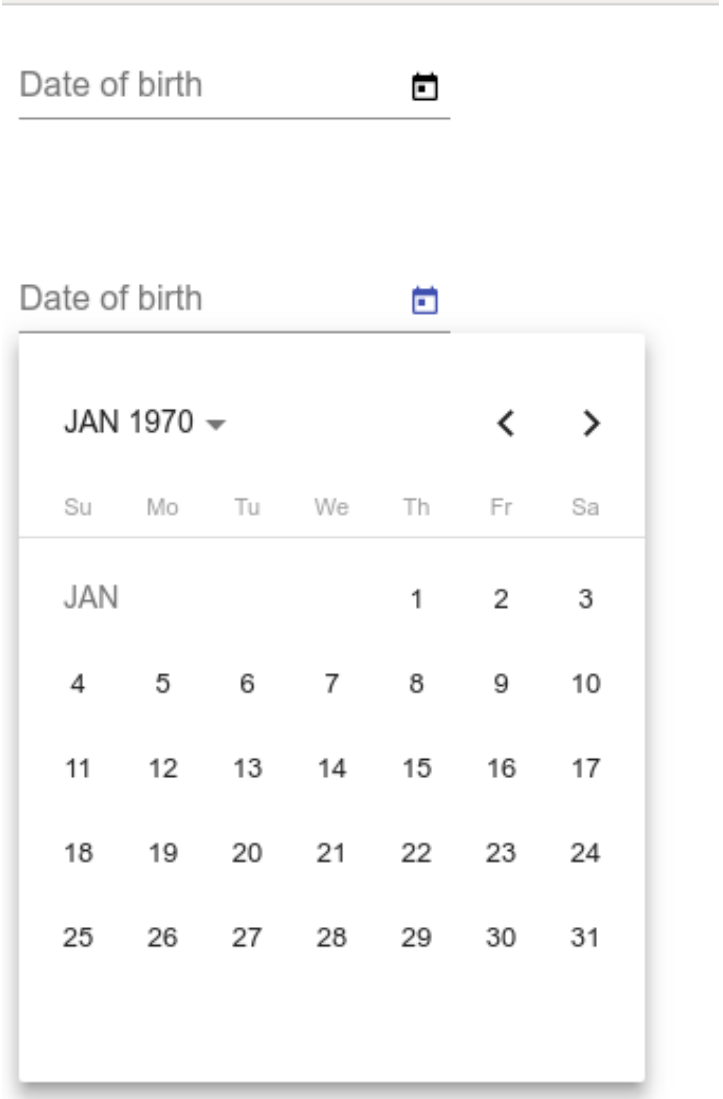
```
npm install --save @angular/material-moment-adapter
```

- Add to app.module.ts

```
import { MatMomentDateModule }
    from '@angular/material-moment-adapter';
```

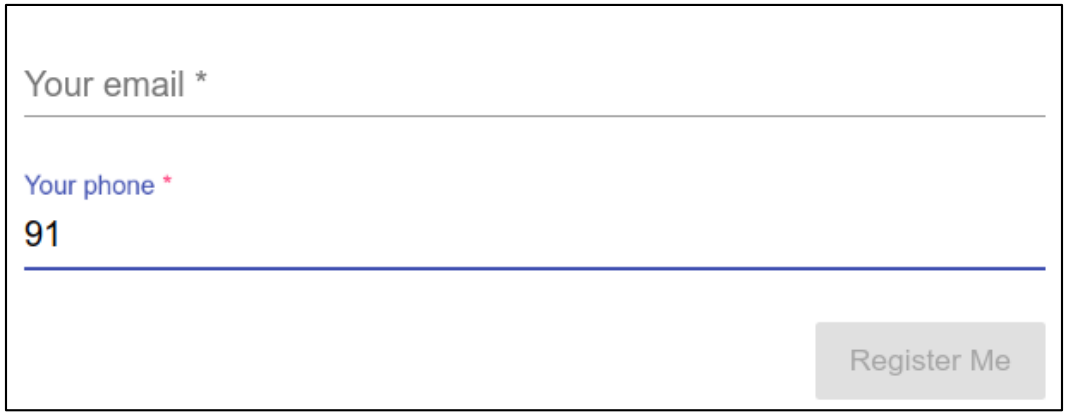
Date Picker

```
<mat-form-field>

  <input matInput
     [matDatepicker]="datepicker"
    placeholder="Date of birth"
    ngModel name="dob">

  <mat-datepicker-toggle matSuffix
     [for]="datepicker">
  </mat-datepicker-toggle>

  <mat-datepicker #datepicker>
  </mat-datepicker>

</mat-form-field>
```



Form Validation

```
<form #form="ngForm" (ngSubmit)="register(form)">
  <mat-form-field>
    <input type="email" name="email"
      placeholder="Your email" matInput
      required email ngModel>
  </mat-form-field>

  <mat-form-field>
    <input type="tel" name="phone"
      placeholder="Your phone" matInput
      required minlength="8" ngModel>
  </mat-form-field>

  <button type="submit" mat-raised-button
      color="primary" [disabled]="form.invalid">
    Register Me
  </button>
</form>`
```



Affordance - Hints and Error Messages

```
<mat-form-field>
  <input type="email" name="email"
     placeholder="Your email" matInput
     #emailField="ngModel"
     required email ngModel>

  <mat-hint *ngIf="emailField.invalid">
    Please enter a valid email
  </mat-hint>

  <mat-error *ngIf="emailField.hasError('email')">
    Invalid email
  </mat-error>

</mat-form-field>
```

Define a template reference to the control

Display the hint if the control is invalid

Display the error if the control's error is because of email validation

Example - Affordance

Your email *

Please enter a valid email

Your phone *

Please enter a 8 digit phone number

Register Me

Your email *

f

Invalid email

Your phone *

Please enter a 8 digit phone number

Register Me

Invalid email format

Your email *

fred@bedrock.com

Your phone *

Please enter a 8 digit phone number

Register Me

Valid input

Defer and Promise

Promise

Customer will get coffee some time in the near future

Deferred

Proprietor prepares the cup of coffee

Resolved

When proprietor signals customer to collect coffee

Promise

- A promise represents a pending value
- Promises can either be
  - resolved – the value is valid and is available
  - reject – the value is not available
- Once a promise has been resolved, in stays resolved
  - Resolution means either the promise is resolve or reject
  - Cannot reset its state, use only once
- Used in JavaScript
  - Prevent blocking because JavaScript is a single threaded environment
  - To coordinate multiple serial or concurrent tasks

Promise - Provider

- Promise object is native to JavaScript
  - Do not need to import any modules to use it
- Pass the promise a callback with 2 parameters
  - The parameters are the resolve and reject function respective

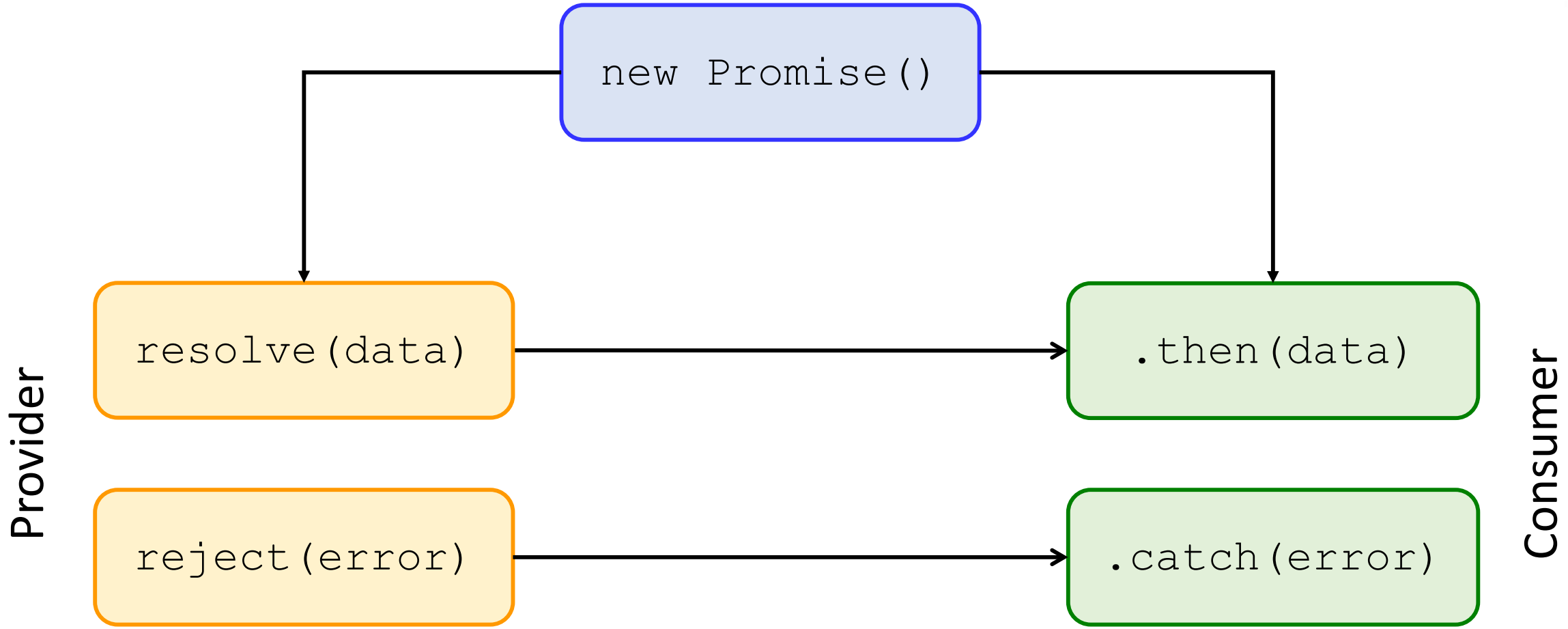
```
const callMe = new Promise((resolve, reject) => {
   //If resolve
   resolve(data);

   //If failed reject
   reject(error);
})
```

Promise - Consumer

- Promise object has 2 functions for listening to resolve and reject
  - Pass a callback
- `then()` for resolve
- `catch()` for reject

```
callMe
  .then((data) => {
    //Promise resolved
  })
  .catch((error) => {
    //Promise rejected
  })
```

Promise


new Promise()
Provider
resolve(data)
reject(error)
.then(data)
.catch(error)
Consumer

Promise Chains

- Any values return from the callbacks of `then()` and `catch()` will be wrapped as promise
- A return from `then()` will resolve to the next `then()`
- Throwing an exception will resolve to the next `catch()`

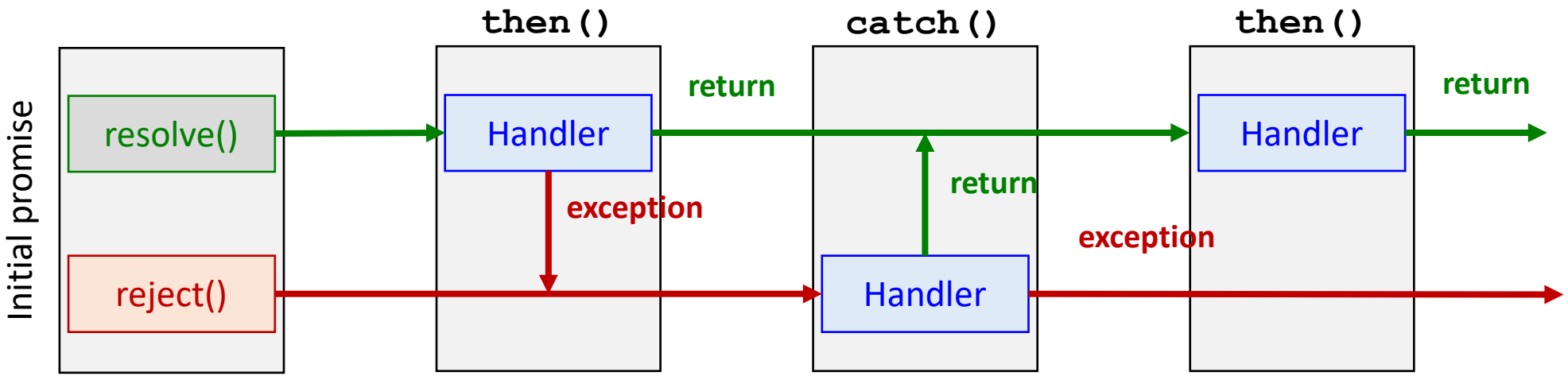


Image inspired from https://speakerdeck.com/danielbrookroberge/asynchronous-pipelines-in-javascript-with-es6-promises

Services

- Services are abstractions for encapsulating reusable code
- Service provides cross-cutting concerns
  - “Horizontal” services like authentication, logging, persistence, etc
- Services are singletons - there is only one instance of the service in the module
  - Provided at the module level
  - Note: not strictly true. Depends on where the service is provided
- Services can access other services thru dependency injection
  - Eg. `HttpClient`
  - Service class must be annotated with @Injectable()

Example - Service

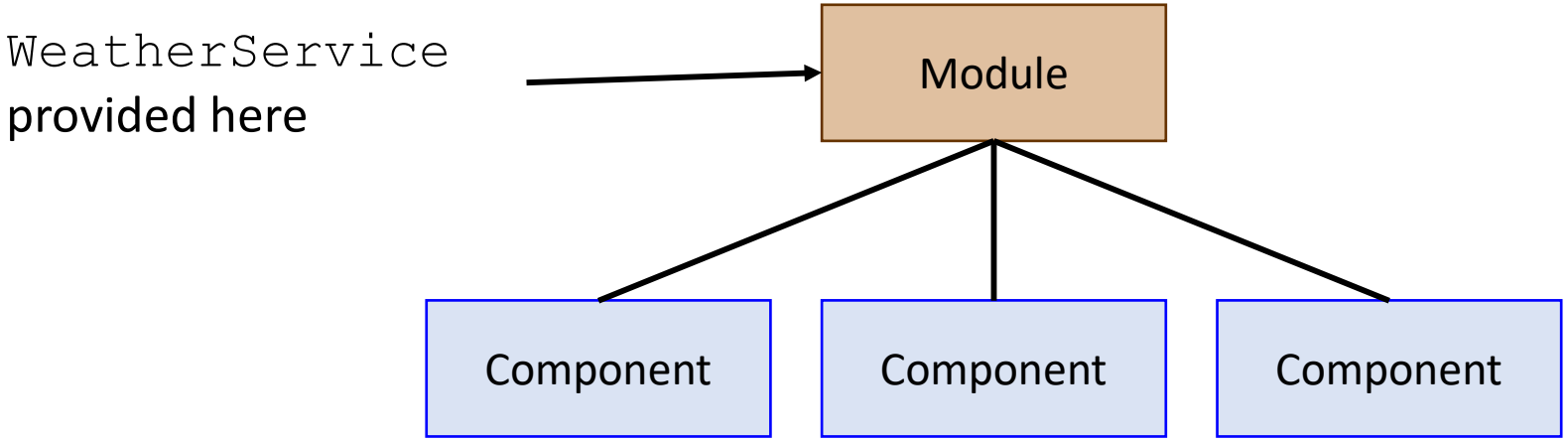
`@Injectable()` is required because we are injecting another service `HttpClient` into `WeatherService`

```
@Injectable()
export class WeatherService {
  constructor(private http: HttpClient) { }

  getWeather(city: string, key: string): Promise<any> {
    const params = new HttpParams()
        .set('q': city)
        .set('appid': key);
    return (
      this.get('http://api.openweathermap.org/data/2.5/weather',
          { params: params })
        .toPromise()
    );
  }
}
```

Example - Service

```
import { WeatherService } from './weather.service';

@NgModule({
  ...
  providers: [ WeatherService ]
})
export class AppModule {
  ...
```

All components and services in a module share the same instance of the service if the service is provided at the module level

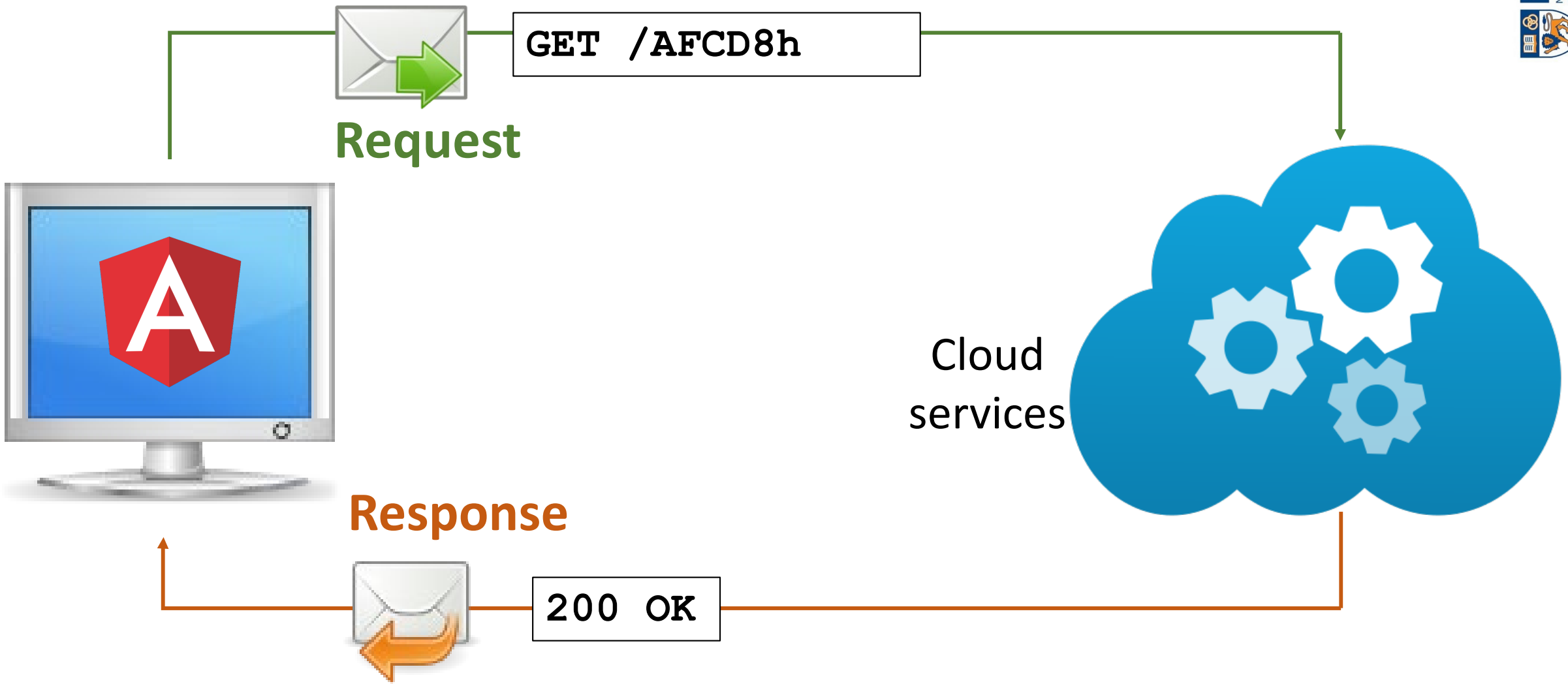


Example - Service

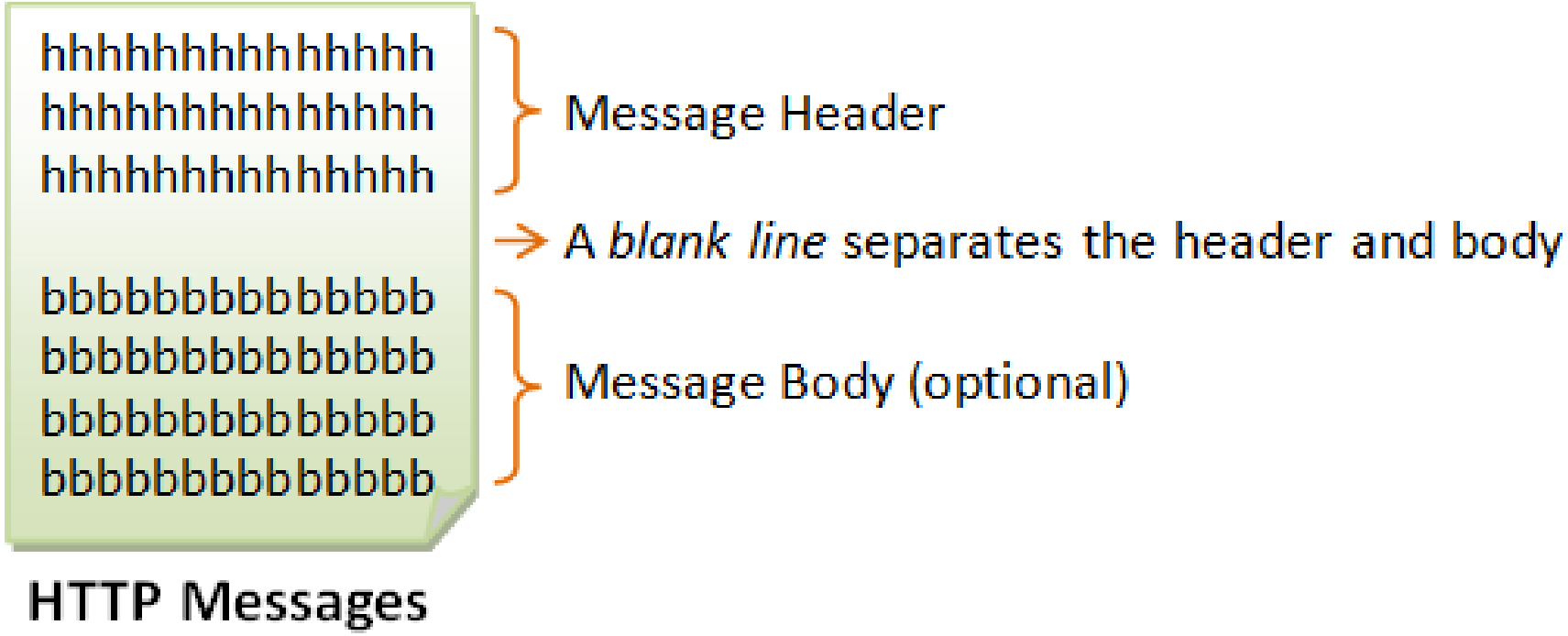
```
import { WeatherService } from ‘./weather.service’;

@Component({ ... })
export class AppComponent {

  constructor(private weatherSvc: WeatherService) { }
  ...
}
```

Once a service has been provided, can be injected into any components in the module

HTTP Request

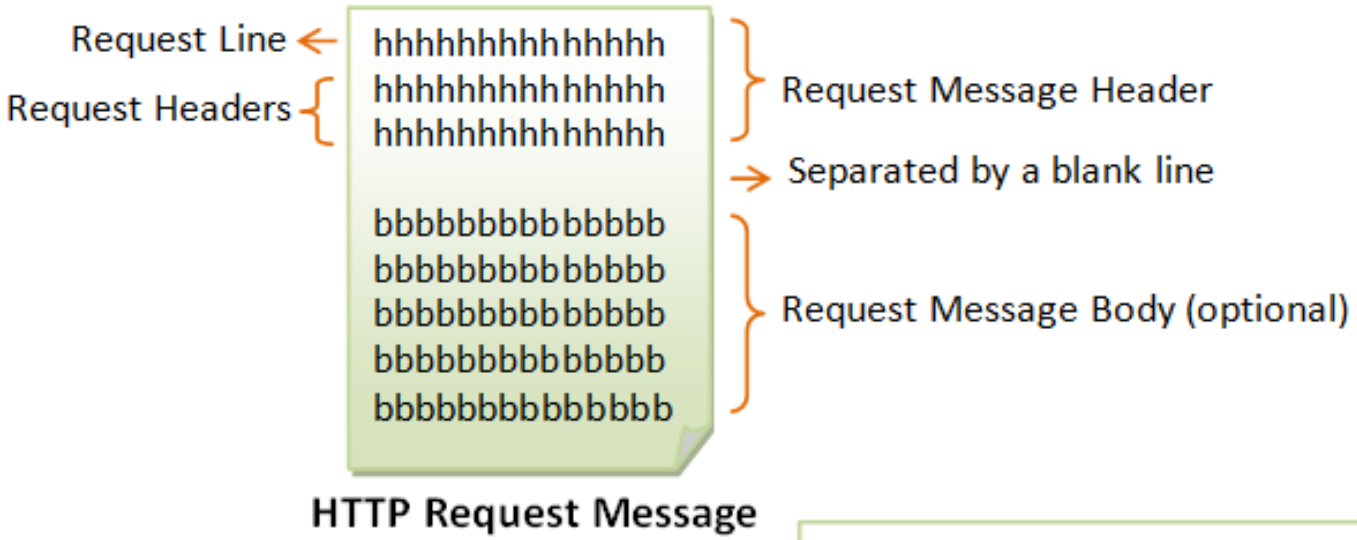


HTTP Message Structure



Images from https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

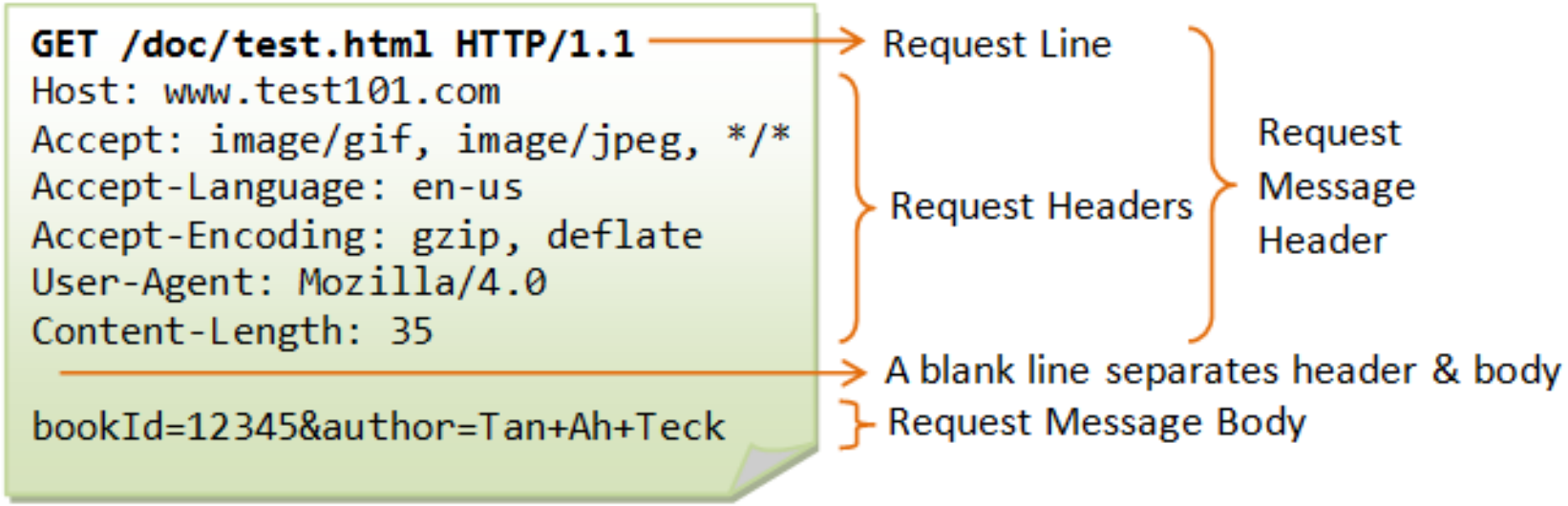
HTTP Request



HTTP method Resource name

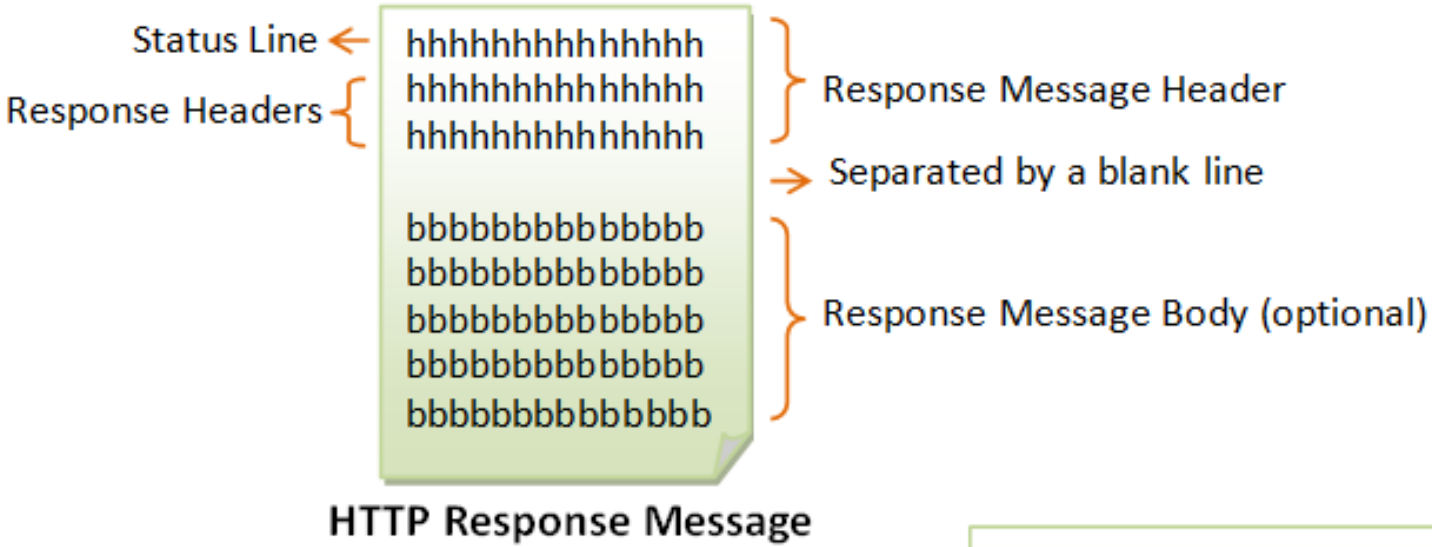
GET /doc/test.html

GET
POST
PUT
DELETE
HEAD
OPTION
TRACE



Images from https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

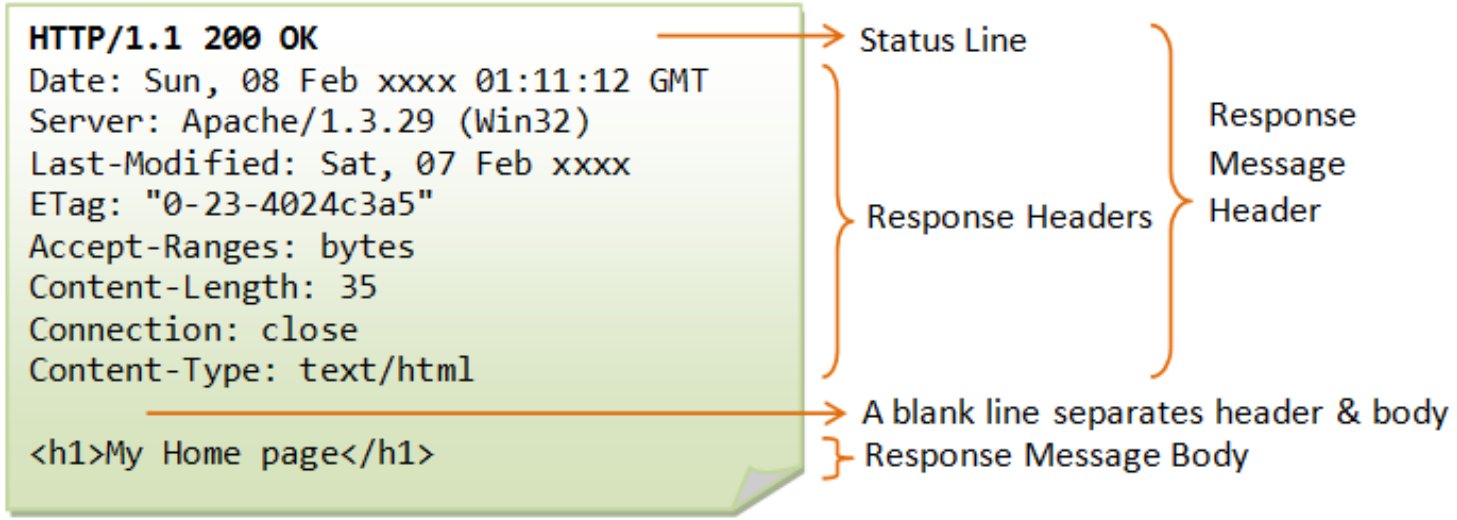
HTTP Response



Status code

200 OK

100
200
300
400
500



Images from https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

HTTP Request Structure

A resource name within the service

GET /customer/1

Verb - what to do

Noun - what to do it to

Method, Resource and Status

Operation	Verb	Noun	Outcome
Read	GET	/customer/1	200 OK
Create	POST	/customer	201 Created
Update	PUT	/customer/1	200 OK
Delete	DELETE	/customer/1	200 OK

REQUEST → ← RESPONSE

HTTPClientModule

- `HTTPClient` is a service available in the http module
- Need to be installed and imported

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

@NgModule({
  imports: [
    HttpClientModule
  ]
})
export class AppModule {
```

HttpClient Service

- The HttpClientModule exports the HttpClient service
- Need to be injected into components or services to be used

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

@Component({ ... })
export class AppComponent {

  constructor(private httpClient: HttpClient) { }
  ..
}
```

Making HTTP Calls

- The **`HttpClient`** is the service for making HTTP request
- **`HttpClient`** provides the following method that maps to its corresponding HTTP method
  - `HttpClient.get(url, configuration)`
  - `HttpClient.post(url, configuration)`
- **`HttpClient`** returns an observable
  - Convert to a promise with `toPromise()`
- **`HttpClient`** assumes all request and response payload are JSON

HTTP Method - GET

- Handling as a Promise

```
this.httpClient.get(url)
  .toPromise()
  .then((data) => {
    for (let i of data) ...
  });
```

Response - an array of the following object

```
{
  name: "fred",
  email: "fred@gmail.com"
}
```

Angular assumes all results are in JSON

HTTP Method - GET

- Handling as a Promise

```
this.httpClient.get(url)
  .toPromise()
  .then((data) => { data.name })
  .catch((error: HttpErrorResponse) => { /* error */ });
```

- `HttpErrorResponse` object has the following properties
  - `status` – status code
  - `error` – error message

HTTP Method - GET

- Type safe response
  - Call the generic version of the `get()` method

```
export interface Customer {
  name: string;
  email: string;
}

this.httpClient.get<Customer[]>(url)
  .toPromise()
  .then((data) => {
     for (let i of data)...
  });
```

Response - an array of the following object

```
{
  name: "fred",
  email: "fred@gmail.com"
}
```

HTTP Method - GET

- Making an invocation with query parameters
  - Create query params with `HttpParams` class

```
HttpParams queryParams = new HttpParams()
   .set("custId", 1234);

this.httpClient.get(url, { params: queryParams } )
  .toPromise()
  .then((data) => {
     data.name
  });
```

Creating a query parameters

Add to the call configuration

`url?custId=1234`

HTTP Method - POST

- **`HttpClient.post`** sends data to as JSON
  - Not as `application/x-www-form-urlencoded`

```
const customer: Customer = {
  name: 'barney',
  email: 'barney@bedrock.com'
}

this.httpClient.post(url, customer)
  .toPromise()
  .then(() => { /* success */ });
```

`customer` as the payload

Angular assumes all content are in JSON

HTTP Method - POST

- **`HttpClient.post`** sends custom headers
  - Not as `application/x-www-form-urlencoded`

```
const jwt = //our token
const headers = new HttpHeaders()
   .set('Authorization', `Bearer ${jwt}`);

this.httpClient.post(url, customer,
   { headers: headers })
  .toPromise()
  .then(() => { /* success */ });
```

Additional headers

Angular assumes all content are in JSON

HTTP Method - POST

- Sending a `x-www-form-urlencoded` payload

```
const customer = new HttpParams()
    .set('name', 'barney')
    .set('email', 'barney@bedrock.com');
```

Construct the playload using `HttpParams` instead of an object

```
const headers = new HttpHeaders()
    .set('Content-Type',
        'application/x-www-form-urlencoded');
```

Set the appropriate content type

```
this.httpClient.post(url,
    customer.toString(),
    { headers: headers })
  .toPromise()
  .then(() => { /* success */ });
```

Call `toString()` to serialize the payload

Single Page Application (SPA)

- SPA are web applications that loads a single HTML called the app shell
- It then dynamically update the contents of app shell as the user interacts with the application
- The app shell is not reloaded when its content is refreshed

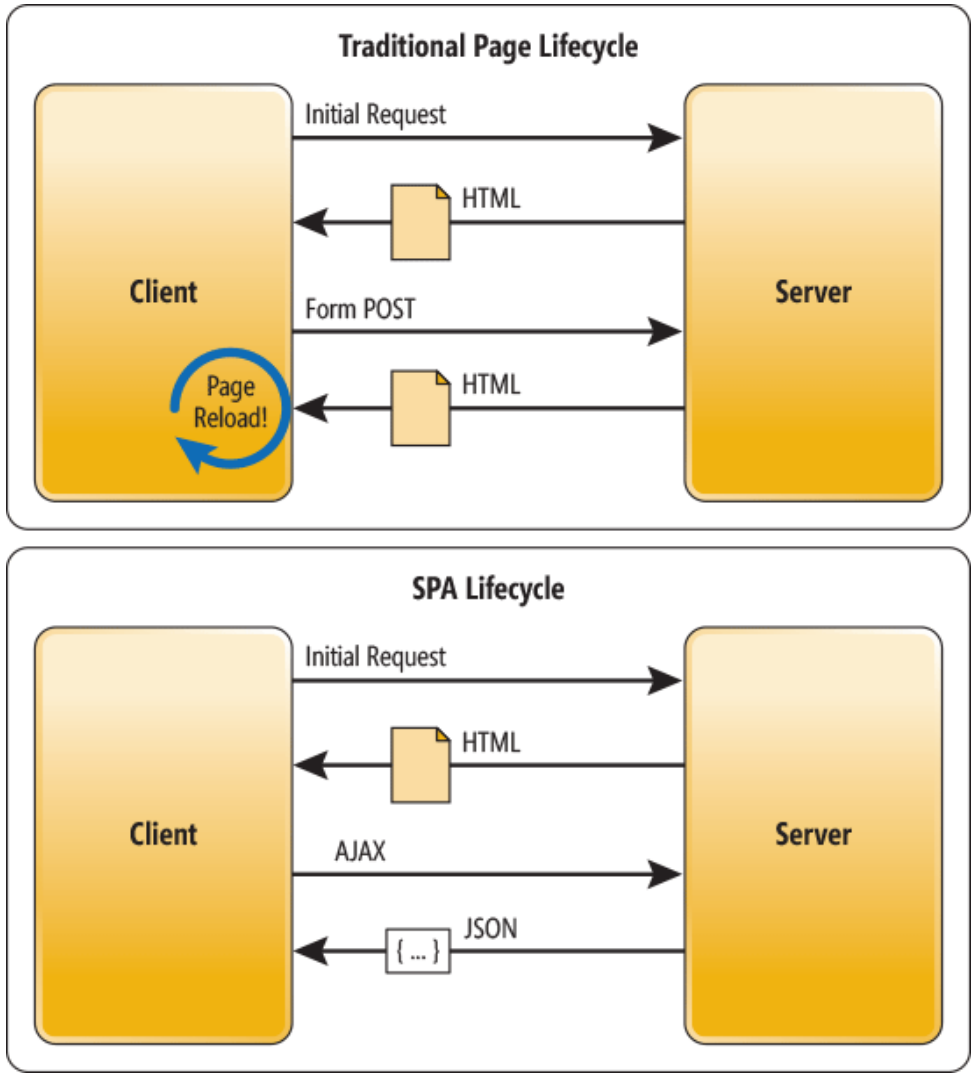
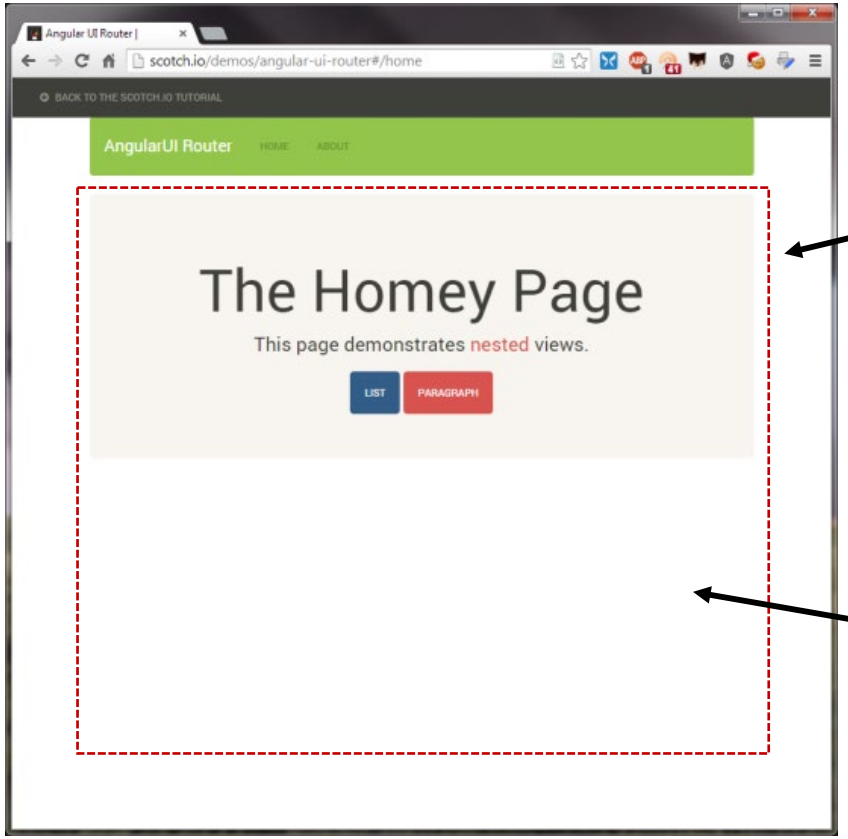


Image from https://msdn.microsoft.com/en-us/magazine/dn463786.aspx

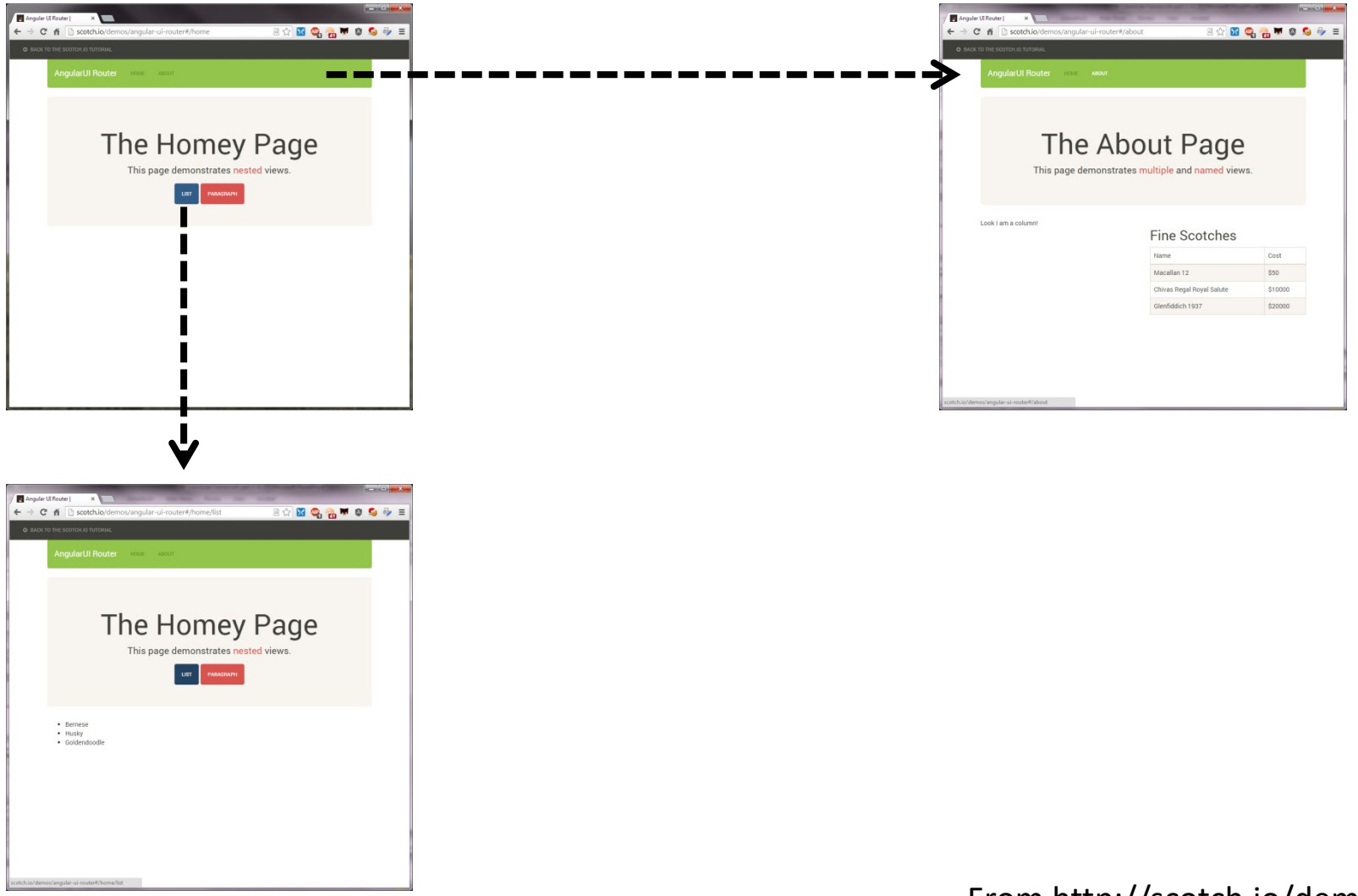
SPA - App Shell



The ‘application shell’ that host the different views/components for different routes.

The component changes depending on the route. This is the outlet
Note the app shell is not reloaded

SPA - Client Side Routing



From http://scotch.io/demos/angular-ui-router

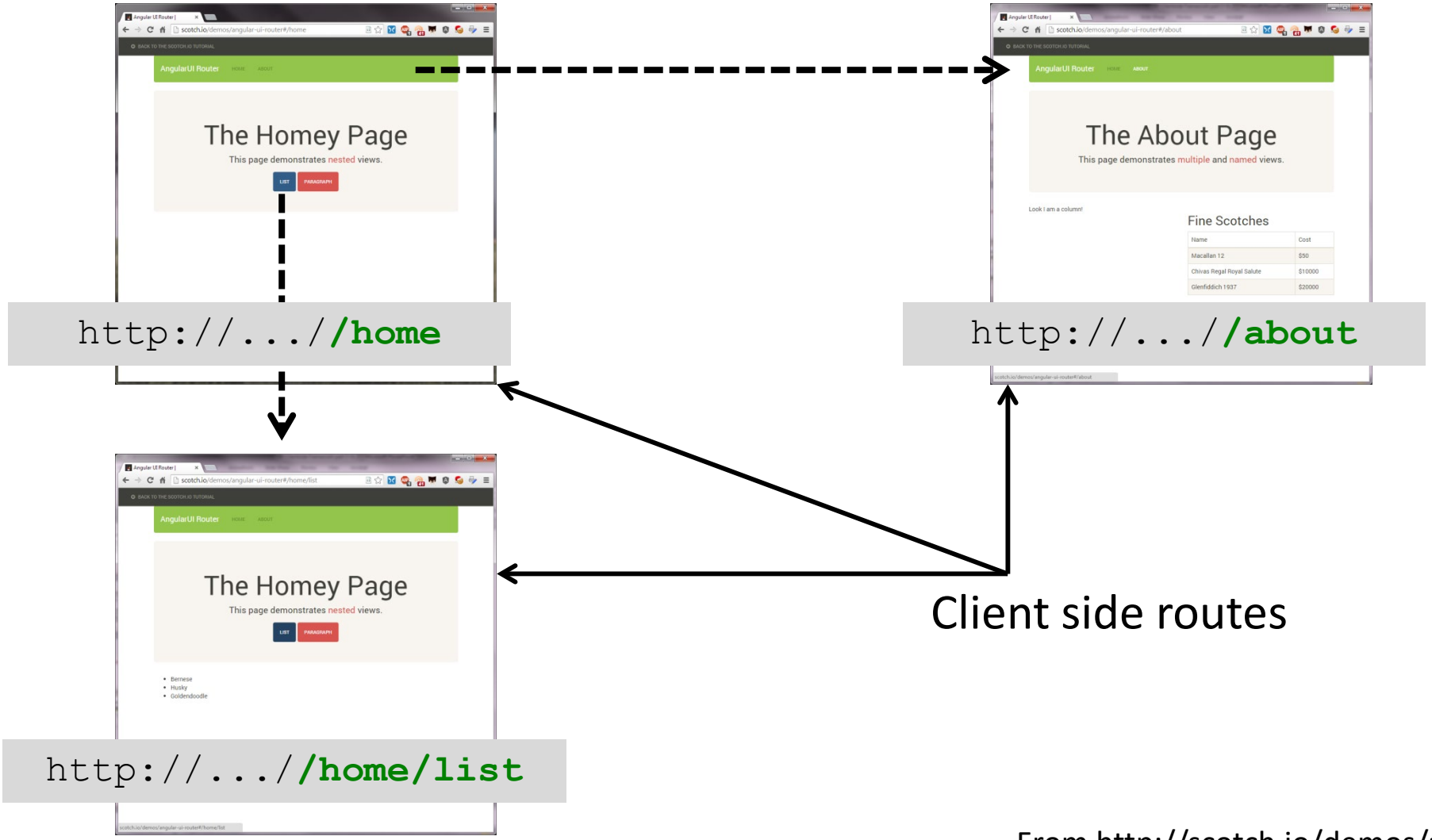
Changing Client Routes

- A change in the client route viz. content of the app shell is triggered by changing the URL
  - Angular monitors the address bar
- `RouterModule` manages the client side routes and maps the URL to a component according to the current URL in the address bar

```
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    ...,
    RouterModule.forRoot({ ... })
  ]
})
```

When loading `RouterModule`, need to define the all the routes in the current application

SPA - Client Side Routing



From http://scotch.io/demos/angular-ui-router

Route

- A route consists of
  - URL
  - Component - this is the component that is loaded when the URL is activated
  - Pre-condition - for activating the route. Optional
  - Post-condition - for leaving the route. Optional

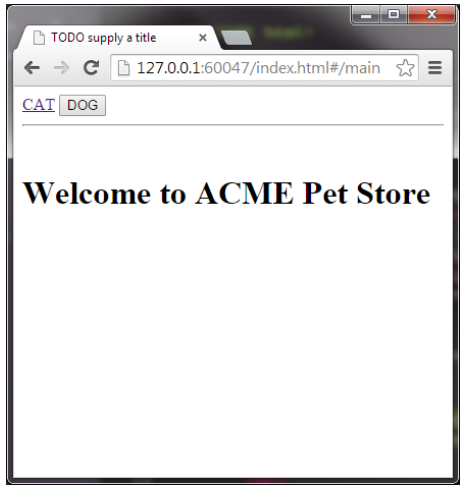
```
const appRoutes: Routes = [
  { path: "dog", component: DogComponent },
  ...
]
RouterModule.forRoot(appRoutes)
```

One route

Path, /dog. Don't include /

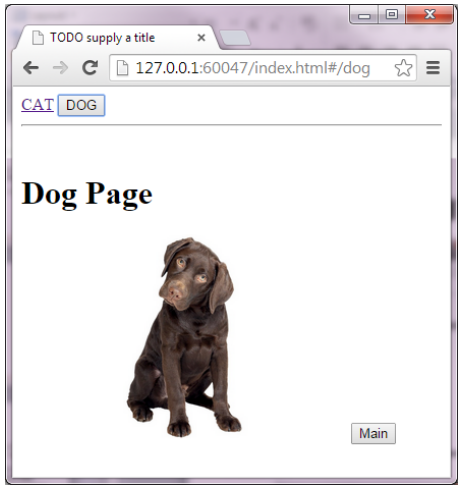
Component to activate

Example - Defining Routes




```
url: main
component: HomeComponent
```

```
url: dog
component: DogComponent
```




```
url: cat
component: CatComponent
```

Example - Defining Routes

```
import { Routes, RouterModule } from '@angular/router';
const appRoutes: Routes = [
     { path: '', component: HomeComponent },
     { path: 'home', component: HomeComponent },
     { path: 'cat', component: CatComponent },
     { path: 'dog', component: DogComponent }
     { path: "**", redirecTo: '/', pathMatch: 'full' }
    ];

@NgModule({
  imports: [
    ...
    RouterModule.forRoot(appRoutes)
  ]
})
export class AppModule {
  ...
```

Same routes

Catch-all route; will be activated when no route matches

Outlet

- Outlet is the location where the components for the routes to display their respective components
- `<router-outlet>` element is used to demarcated where in the app shell should the component be displayed
- A typical place to is `app.component.html`

Router Outlet

app.component.html

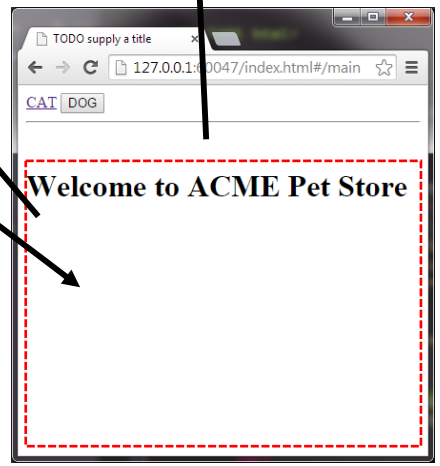
```
<div>
  <router-outlet>    </router-outlet>
</div>
```

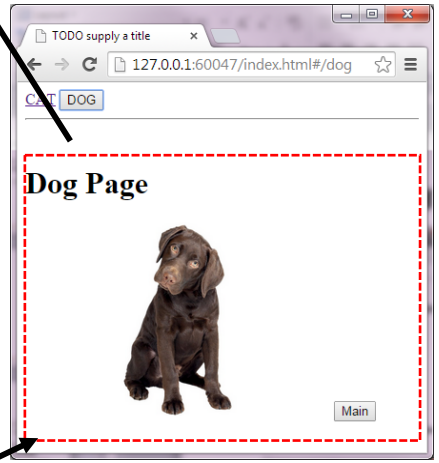
home.component.html

```
<h1>Welcome to ACME Pet Store</h1>
```

dog.component.html

```
<h1>Dog Page</h1>
<img src="images/dog.jpg">
<button type="button">
  Back
</button>
```





Changing Client Side Route

- A change of client's view is triggered by changing the URL
- URL can be changed by
  - User interactivity - eg. user clicking on a link
  - Programmatically with TypeScript
- Use `routerLink` directive instead of `href` attribute in `<a>`

```
<a [routerLink]="['/home']">Home<a>

  const appRoutes: Routes = [
      { path: '', component: HomeComponent },
      { path: 'home', component: HomeComponent },
      ...
    ];
```

Clicking the link will trigger the `HomeComponent` to be display at the router outlet

Changing Client Side Route Programmatically

- `Router` service exposes methods to programmatically to change routes
  - Router service is provided by `RouterModule`
  - Inject into your component
- Use `navigate()` method
  - Parameters is exactly the same as `routerLink` directive

```
this.router.navigate([‘/home’]);
```

`router` is of type `Router`.
Injected into component

Changing Client Side Route Programmatically

app.component.ts

```
import { Router } from '@angular/router';
@Component({
  ...
})
export class DogComponent {

  constructor(private router: Router } { }

  toDog() {
   this.router.navigate(['/dog']);
  }
}
```

Inject `router` service into component

Click event handler

Client Side Routes as Application States

- Routes can be used to represent the current state of the application
  - Eg. `/customer/3` might represent the customer with customer id `3` is current being displayed
- Parts of the route is then static and are dynamic
  - Eg. `/customer/1`, `/customer/2`, `/customer/3`
- Think of the dynamic portion as ‘parameters’

Parameterizing Routes

- Routes are parameterized with a colon (:)

/customer/1
/customer/2
/customer/3

Trigger `CustomerComponent` whenever these URLs are activated

`CustomerDetailComponent`

```
const appRoutes: Routes = [
  { path: 'customer/:custId', component: CustomerDetailComponent
},
  ...
]
```

Route parameter - the variable part of the route captured in `:custId`

Retrieving the Route Parameter

- Use `ActivatedRoute` to retrieve the route parameter
  - `ActivatedRoute` is a service from `RouterModule`

```
@Component({ ... })
export class CustomerDetailComponent implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    const custId = this.activatedRoute.snapshot.params.custId;
    //...
  }
}
```

Retrieve the route parameter

Activating Parameterized Route

- By clicking

```
<a *ngFor="let c of customers"
   [routerLink]="[ '/customer', c.custId ]">
   {{ c.name }}
</a>
```

- Programmatically

```
this.router.navigate(['/customer', custId ]);
```

Parameter Routes Illustrated

```
<a [routerLink]="['/customer', 1]>Jumbo Eagle Corp</a>

                    RouterModule.forRoot([
                      ...
                      { path: '/customer/:custId',
                          component: CustomerDetailsComponent }
                    ])


                          ActivatedRoute.snapshot.params.custId

          @Component({...})
          export class CustomerDetailComponent
             implements OnInit {
           constructor(private activatedRoute: ActivatedRoute,) { }
```

Material List

Star Wars

Luke Skywalker

C-3PO

R2-D2

Darth Vader

Leia Organa

Owen Lars

Beru Whitesun lars

R5-D4

Biggs Darklighter

Obi-Wan Kenobi

PeopleComponent

/films/1
/films/2
/films/3

People id

Star Wars

The Phantom Menace

Episode: 1

Turmoil has engulfed the Galactic Republic. The taxation of trade routes to outlying star systems is in dispute. Hoping to resolve the matter with a blockade of deadly battleships, the greedy Trade Federation has stopped all shipping to the small planet of Naboo. While the Congress of the Republic endlessly debates this alarming chain of events, the Supreme Chancellor has secretly dispatched two Jedi Knights, the guardians of peace and justice in the galaxy, to settle the conflict....

Attack of the Clones

Episode: 2

There is unrest in the Galactic Senate. Several thousand solar systems have declared their intentions to leave the Republic. This separatist movement, under the leadership of the mysterious Count Dooku, has made it difficult for the limited number of Jedi Knights to maintain peace and order in the galaxy. Senator Amidala, the former Queen of Naboo, is returning to the Galactic Senate to vote on the critical issue of creating an ARMY OF THE REPUBLIC to assist the overwhelmed Jedi....

FilmComponent

Material List

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      { path: '', component: PeopleComponent },
      { path: '/people', component: PeopleComponent },
      { path: 'films/:pId', component: FilmComponent }
    ])
  ]
})
export class AppModule {
```

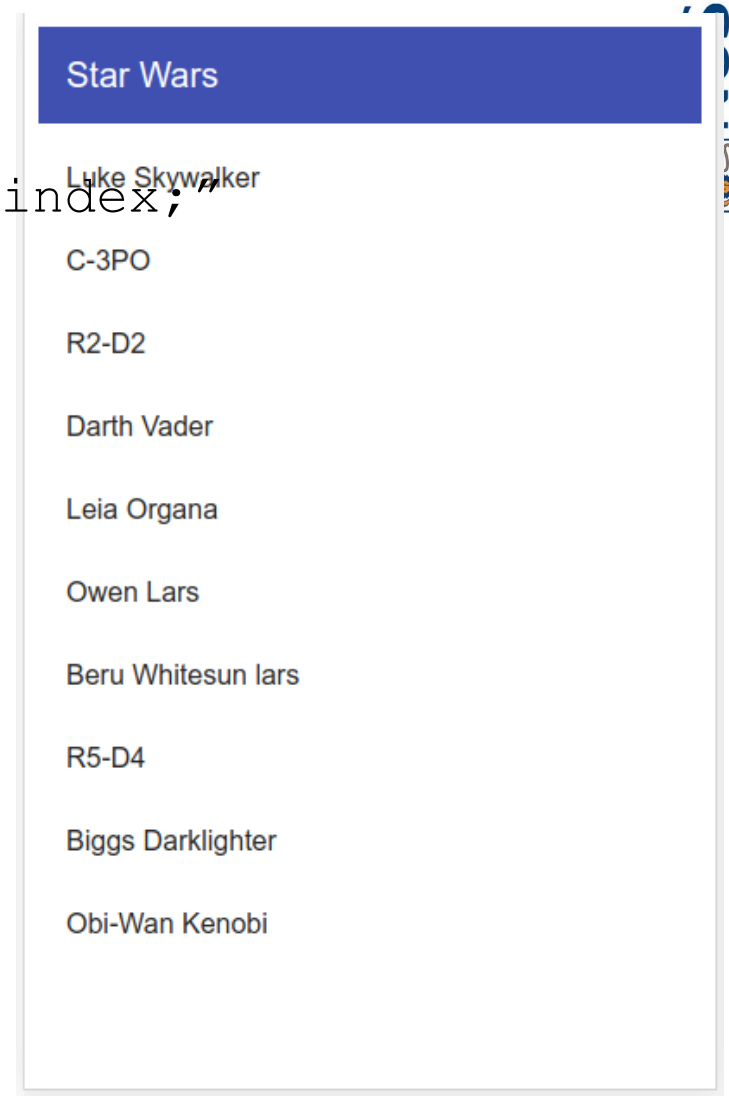
Material List - PeopleComponent

```
<mat-nav-list>
   <mat-list-item *ngFor="let p of people; let i = index;"
         (click)="getFilms(i)">
      {{ p.name }}
   </mat-list-item>
</mat-nav-list>
```

Get all the film ids that the character appeared in; save this list in an array

```
getFilms(idx) {
   const films = //an array of films by idx
   this.router.navigate(['/films', idx],
      { queryParams: { fids: films.join('|') });
}
```

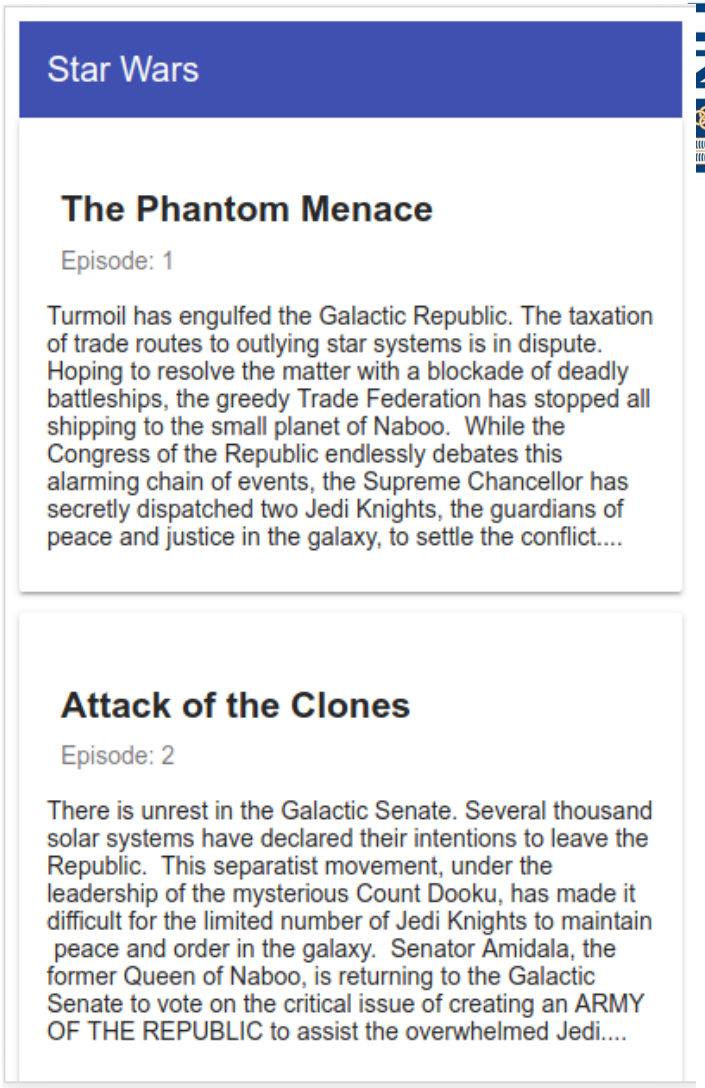
Join all the film ids to a string delimited by |. Pass these over to the next route query parameter



Material List

/films/10?fids=2|5|4|6|3|1

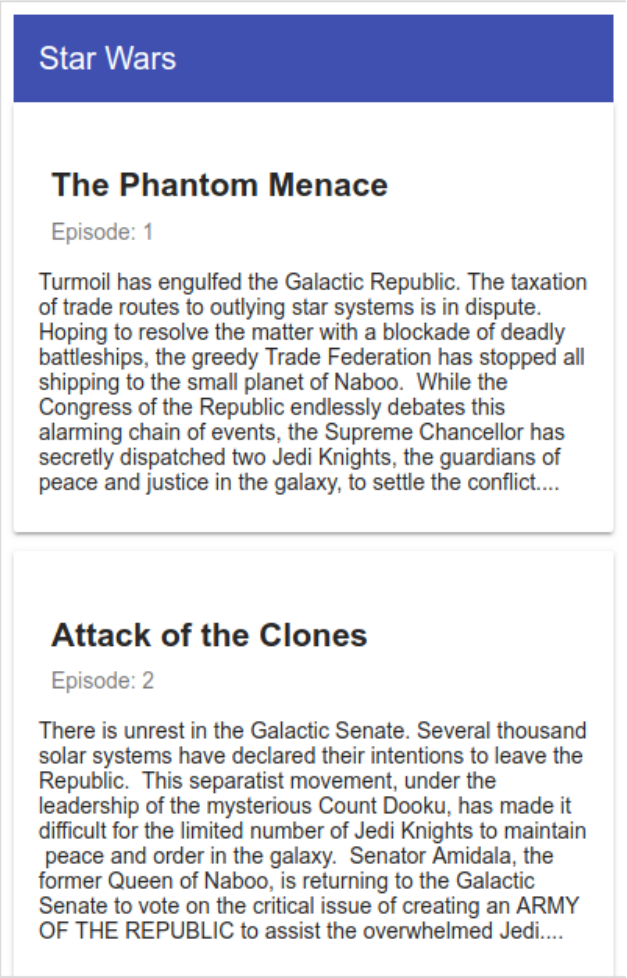
Additional parameters can be passed as query parameter from on route to the next



Material List

```
<div fxLayout="column" fxLayoutGap="2vh">
    <mat-card *ngFor="let f of films">
      <mat-card-header>
         <mat-card-title>
            <h2>{{ f.title }} </h2>
         </mat-card-title>
         <mat-card-subtitle>
            {{ f.episode_id }}
         </mat-card-subtitle>
         <mat-card-content>
            <p>{{ f.opening_crawl }}</p>
         </mat-card-content>
    </mat-card>
</div>
```

```
ngOnInit() {
   const pId = this.activatedRoute.snapshot.params.pId;
   const fids = this.activatedRoute.snapshot.queryParams
                        .fids.split('|');
   this.films = //get the films
}
```



Mobile Application

Native

- Access native API
- Cannot run on multiple platform
- Run offline
- Distribute via app store

Web

- Cannot access native API
- Multiple platform support
- Cannot run offline
- Distribute via the web

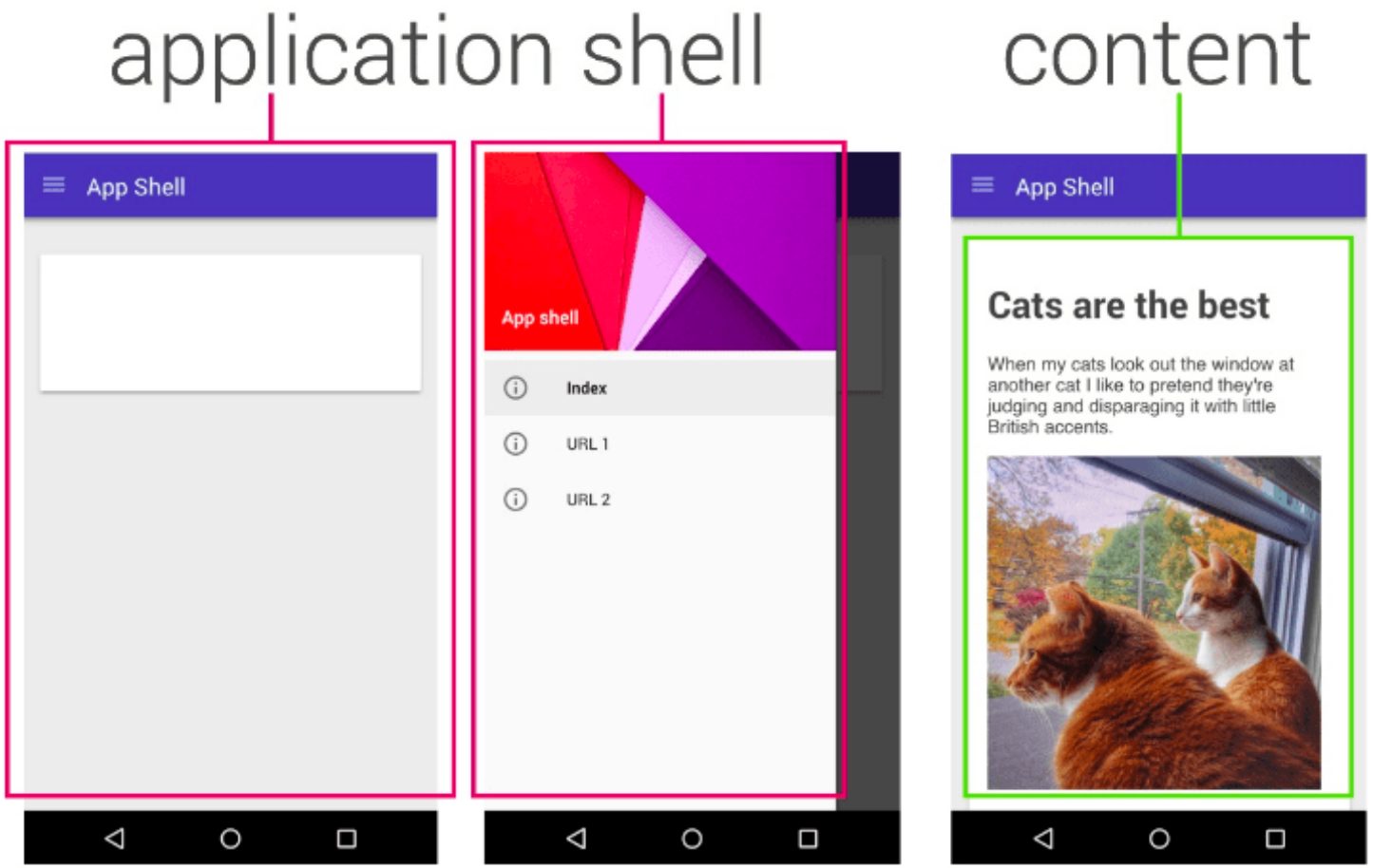
Hybrid

- Can access native API
- Multiple platform support
- Run offline
- Distribute via the web

Progressive

- Limited by the browser
- Multiple platform support
- Run offline
- Distribute via the web

Progressive Web Application



Cached shell loads **instantly** on repeat visits.

Dynamic content then populates the view

Image from https://developers.google.com/web/fundamentals/architecture/app-shell

Application Manifest

- A JSON file that provides information about an application
  - Eg. name, icon, splash screen, screen orientation, etc.
- The purpose of the manifest is to allow web application to be installed on a mobile device's home screen
  - If supported
- Is a core building block of PWA
  - Service worker
  - Push notification
  - Offline cache

Application Manifest

- Create a JSON file with one or more of the following properties
  - `name` - application's name
  - `short_name` - for use in space constrained
  - `start_url` - the URL that start/bootstraps the application
  - `scope` - what are the pages to be included with this application; typical value is `/`
  - `display` - how the application should be displayed. Valid values are
    - `standalone` - no browser control
    - `fullscreen` - similar to standalone
  - `background_color` - RGB
  - `theme_color` - top bar calor RGB
  - `icons` - an array of icons
  - `orientation` - valid values include
    - `landscape`, `portrait`, `any`

Example - manifest.json

```
{
  "name": "My Awesome App",
  "short_name": "AwApp",
  "theme_color": "#2196f3",
  "background_color": "#2196f3",
  "display": "standalone",
  "scope": "/",
  "start_url": "/",
  "icons": [
    {
      "src": "images/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png"
    },...
  ]
}
```

Application Manifest Generator

https://app-manifest.firebaseapp.com/

Unzip in your application's `src` directory

```
src
├── app
│   ├── app.component.css
│   ├── app.component.html
│   ├── app.component.spec.ts
│   ├── app.component.ts
│   ├── app.module.ts
│   └── material.module.ts
├── assets
├── environments
│   ├── environment.prod.ts
│   └── environment.ts
├── favicon.ico
├── images
│   └── icons
│       ├── icon-128x128.png
│       ├── icon-144x144.png
│       ├── icon-152x152.png
│       ├── icon-192x192.png
│       ├── icon-384x384.png
│       ├── icon-512x512.png
│       ├── icon-72x72.png
│       └── icon-96x96.png
├── index.html
├── main.ts
├── manifest.json
├── polyfills.ts
├── styles.css
├── test.ts
├── tsconfig.app.json
├── tsconfig.spec.json
└── typings.d.ts
```

Add Application Manifest to Application

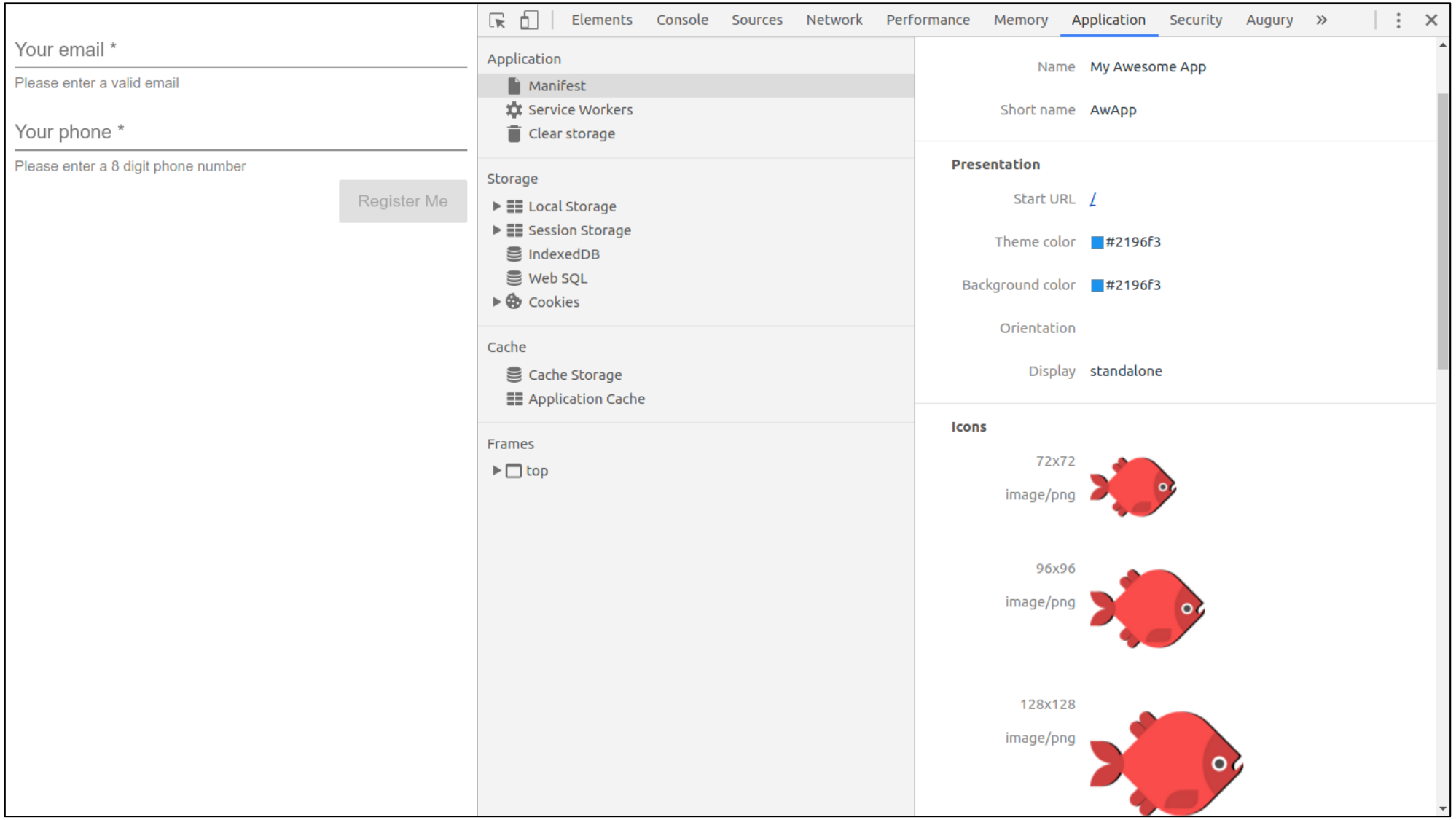
- Add `<link>` to `src/index.html`

```
<link rel="manifest" href="/manifest.json">
```

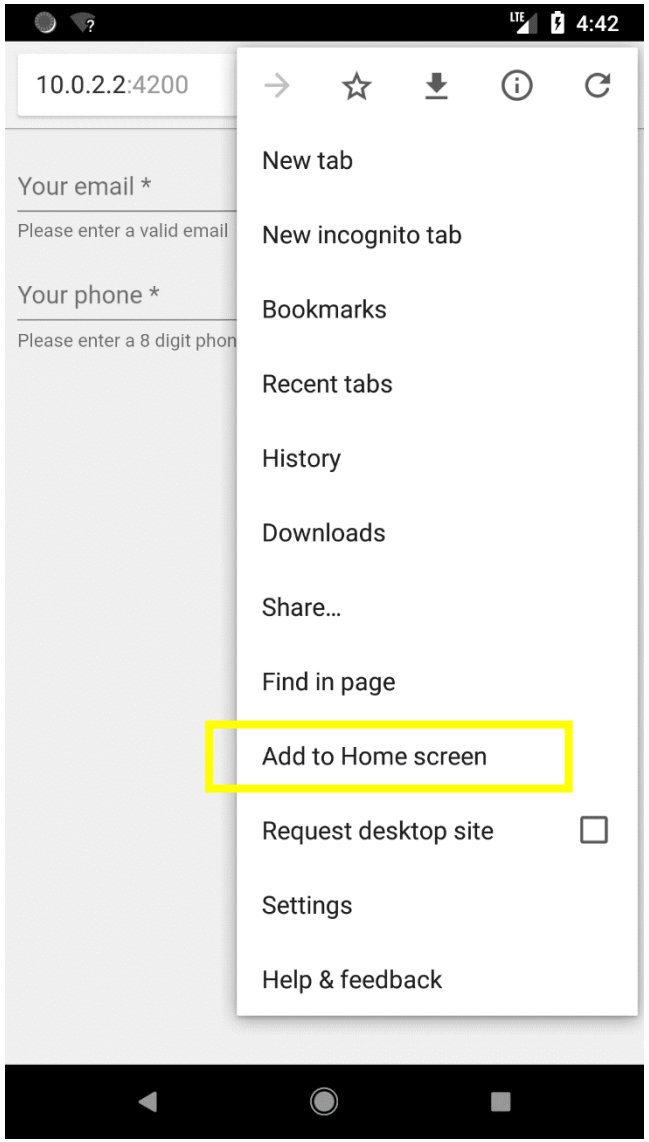
- Include `manifest.json` and `images` directory as part of the build
  - Edit `angular.json`

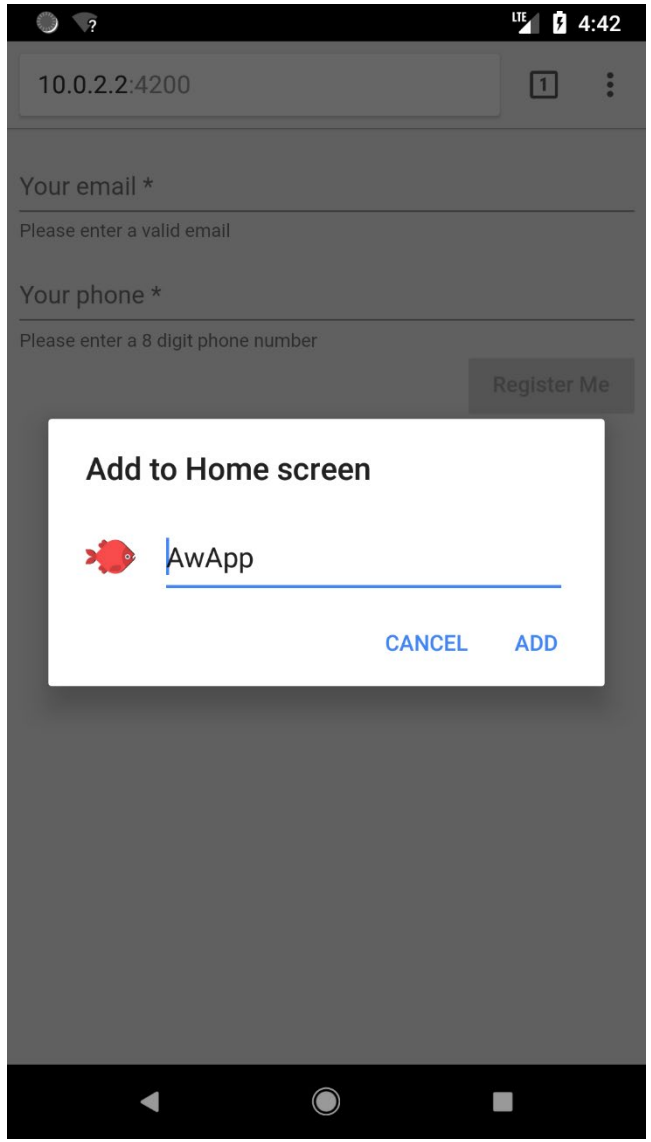
```
"apps": [
  {
    "assets": [ "assets", "favico.ico",
      "manifest.json", "images"
    ],
]
```

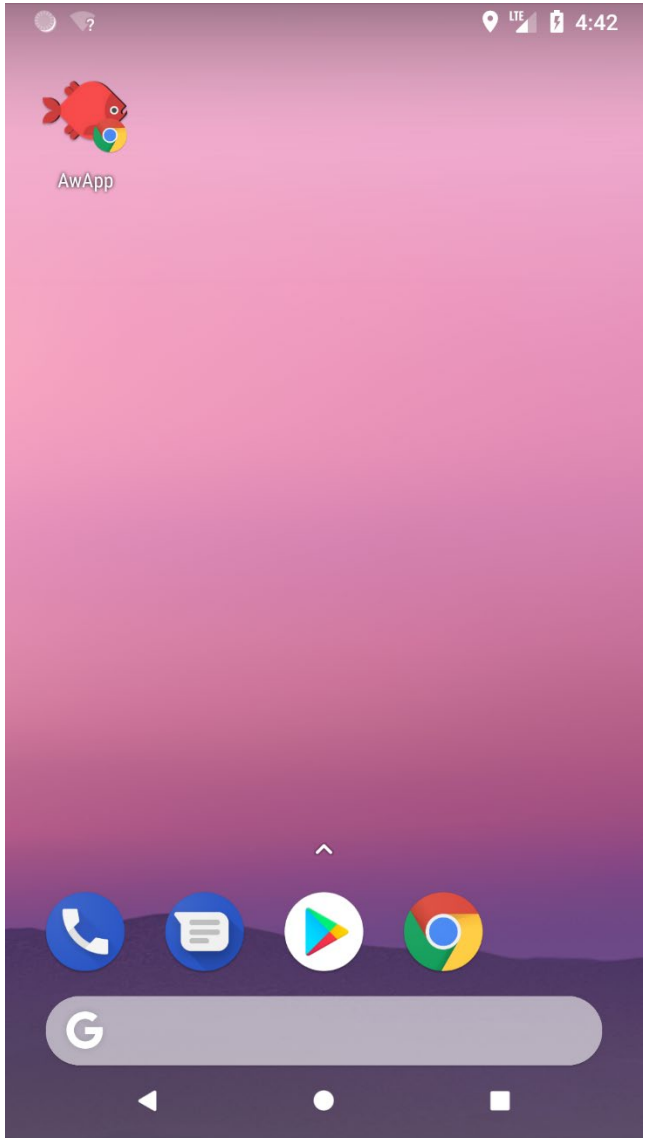
Application Manifest Installed



On Mobile Phone







Publishing to GitHub Pages

- GitHub Pages is a web hosting website for static pages
  - There are limitations
  - See https://help.github.com/articles/what-is-github-pages/#usage-limits
- Accessed with the following

  `https://<username>.github.io/<repo_name>`

- GitHub will serve pages from **`gh-pages`** branch of your repository

Setup

- Install angular-cli-ghpages

```
sudo npm install -g angular-cli-ghpages
```

- Ensure that your Angular application is clean viz. committed
- Include your login name as part of your repo (origin) URL
  - Otherwise you will have to enter your username and password during the deployment

```
git remote add origin https://<usrename>@github.com/<username>/<repo_name>.git
```

Deploying Angular Applications to GitHub Pages

Use the GitHub Pages as the root

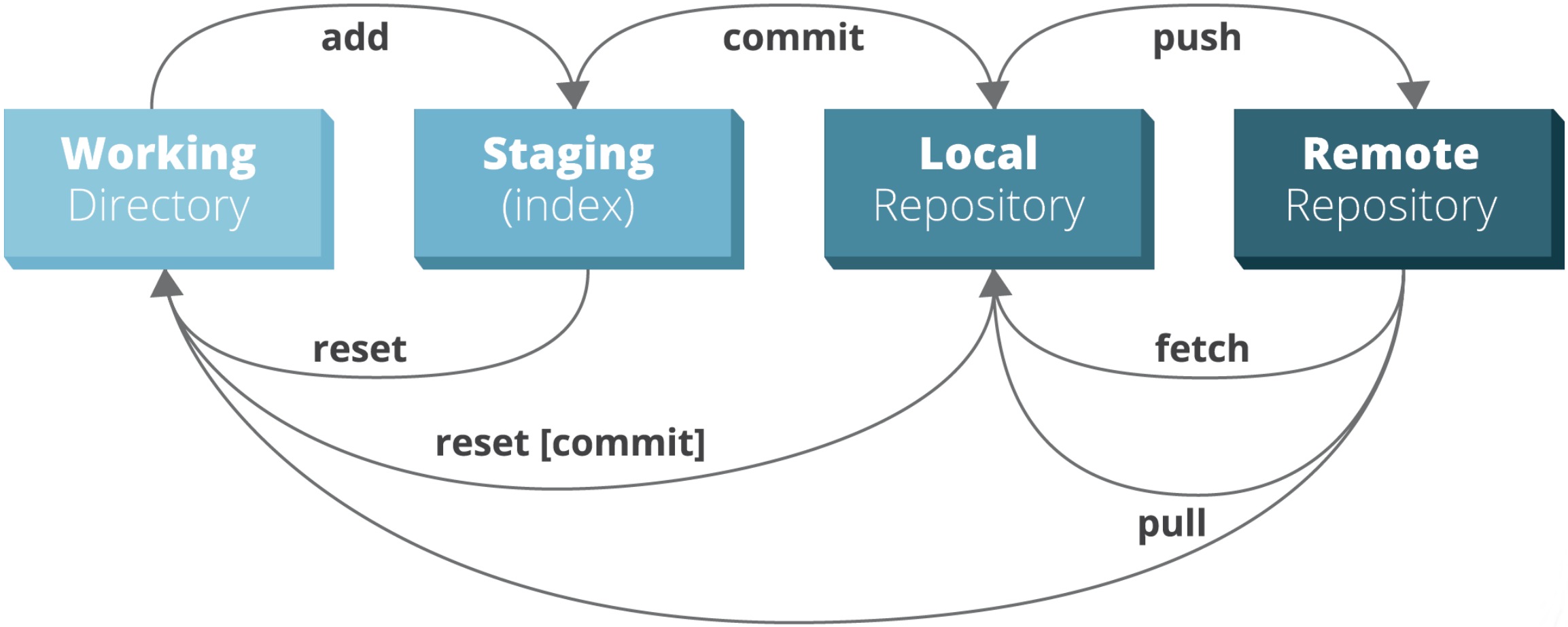
```
ng build --prod \
  --base-href "https://<username>.github.io/<repo_name>"
```

`ngh` (or `angular-cli-pages`)

Build and push application to GitHub

Appendix

Git Workflow



Git Commands

- Initialize a directory as a Git repository
  - Not required if project is generated by Angular CLI

```
git init
```

- Add files to the staging area

```
git add .
```

- Commit files to the local repository

```
git commit -m 'commit message'
```

Git Commands

- Push local repository to remote repository

```
git push -u origin master
```

- Adding a remote repository

```
git remote add origin <git repo URL>
```

- Syncing local repo with remote

```
git pull origin master
```