Angular
Forms, Routing, Services and HttpClient

Outline

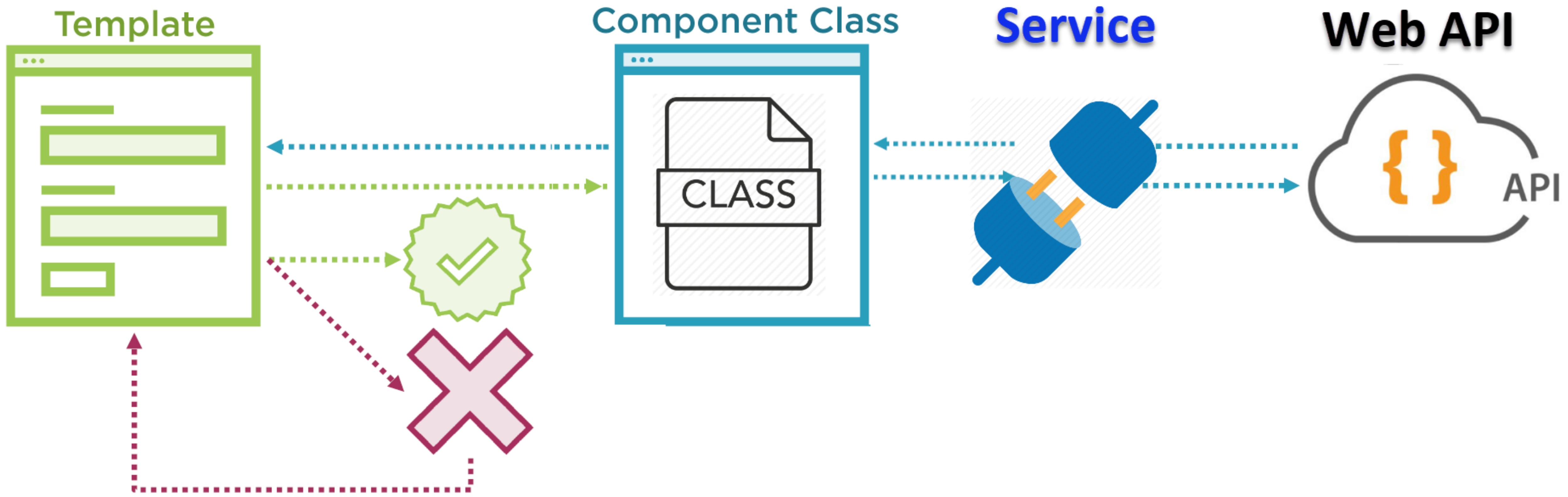
- Forms
- HttpClient
- Routing
- Services
- Angular CLI

Forms

- Template-driven forms
- Reactive forms
- Form validation

Back

Angular Forms



Tracks form and input elements **Value** & **Validation state**

- Provides **model data** for the template

- Handles

User interactions

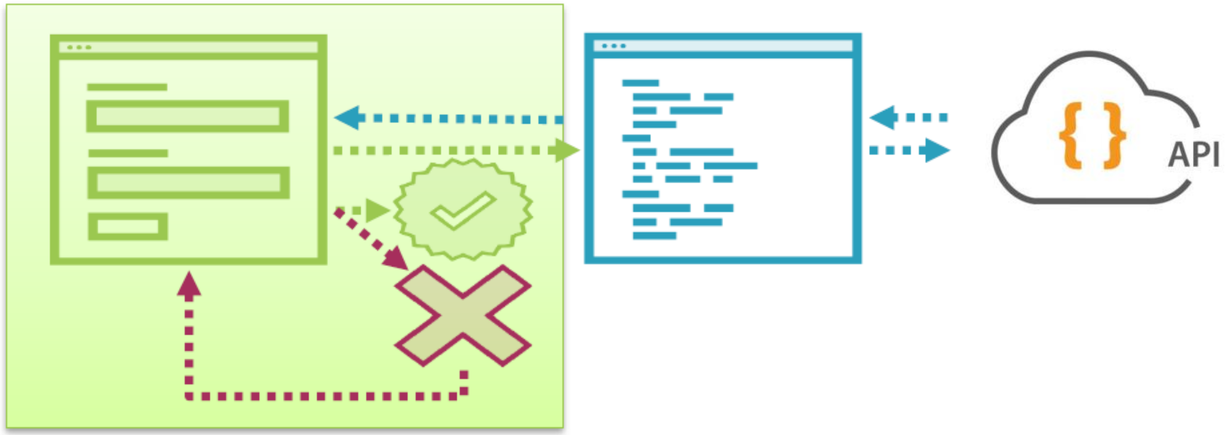
Events

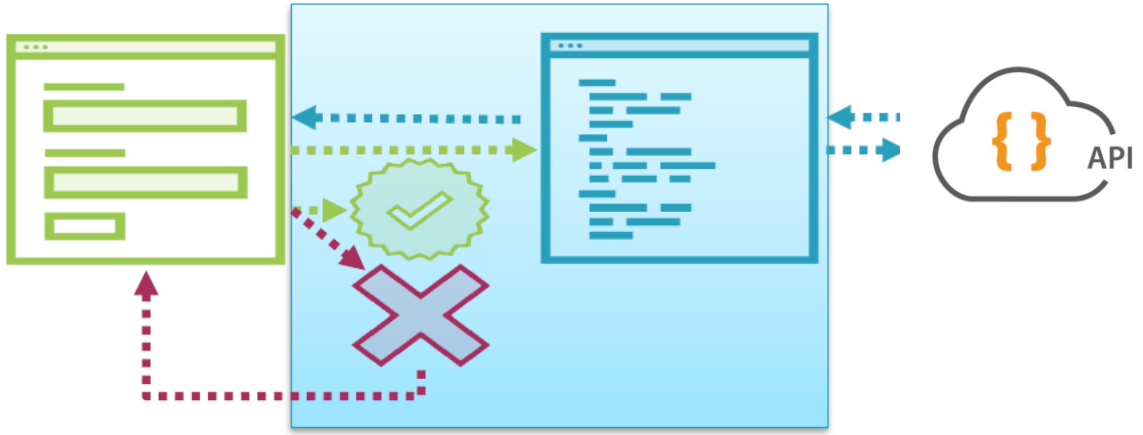
Data Access: interacts with backend Web API

Provides access to backend data and functionality

Template-driven Forms

Reactive Forms





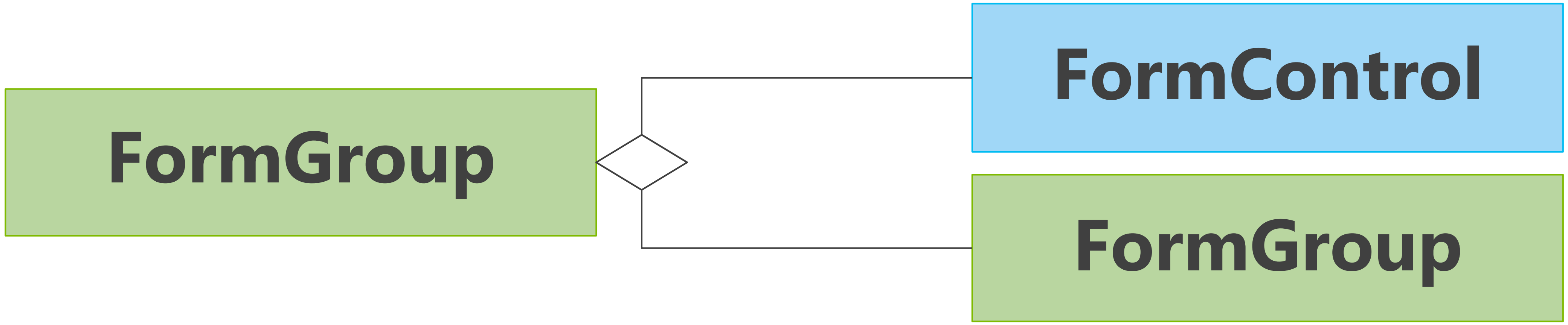
- Most of the form creation and configuration in the Template
- Two-way data binding -> Minimal component code
- Automatically tracks form and input element state
- Easy to use

- More flexible and handles more complex scenarios
- Easier to perform an action on a value change (i.e., React to input value/status changes)
- Easily add input elements dynamically
- Easier unit testing

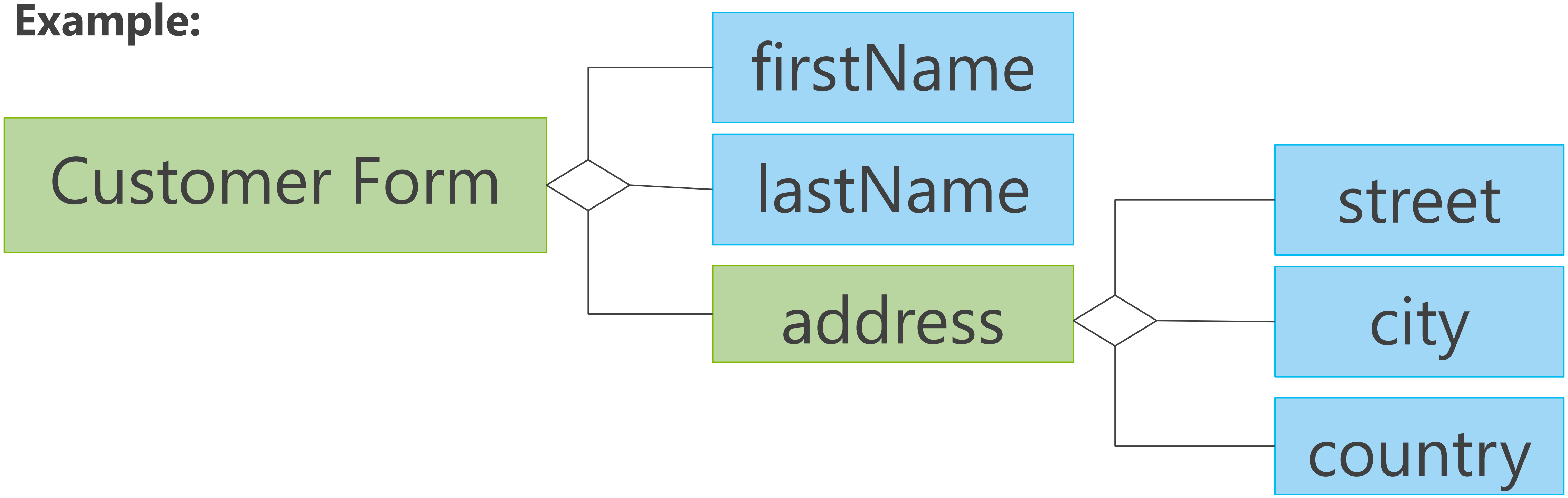
Example Complex Scenarios that can be handled using Reactive Forms

- Dynamically add input elements:
  - Add multiples addresses to a Customer
  - Add multiple items to an order
- Different validation for different situations
  - Make the phone number required if the customer selects notification by SMS
- Watch what the user types
  - Search and fill the products dropdowns as the user types

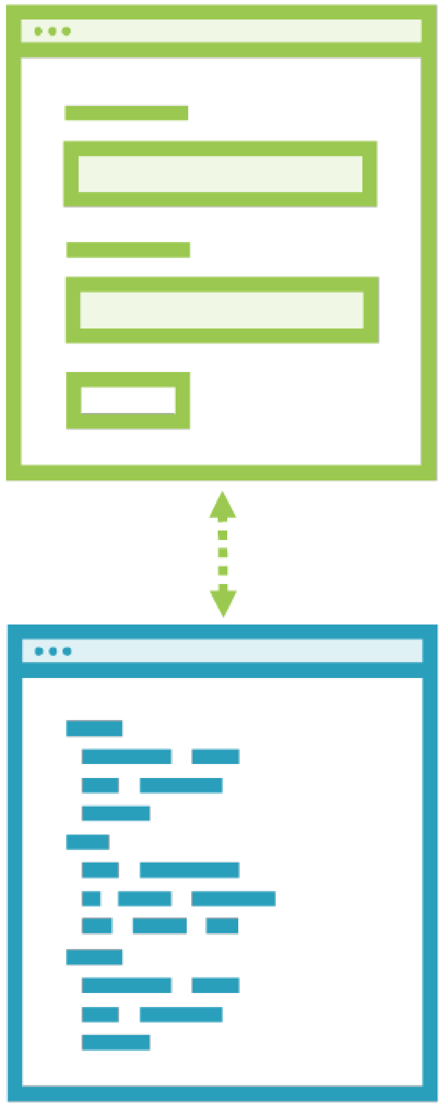
Form Building Blocks

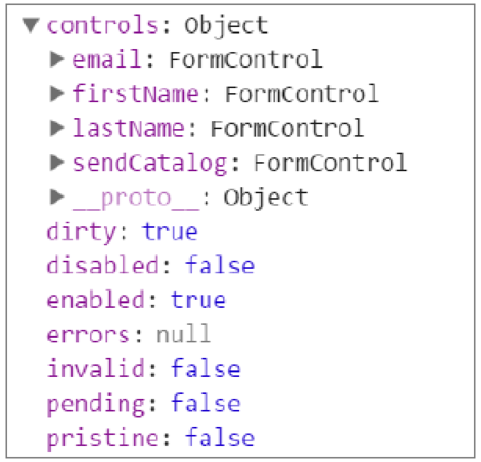


Example:



Template-driven Forms





```
▼ controls: Object
  ▶ email: FormControl
  ▶ firstName: FormControl
  ▶ lastName: FormControl
  ▶ sendCatalog: FormControl
  ▶ __proto__: Object
  dirty: true
  disabled: false
  enabled: true
  errors: null
  invalid: false
  pending: false
  pristine: false
```

Template:

- Form element
- Input element(s)
- Data binding
- Validation rules (attributes)
- Validation error messages
- Form model automatically generated

Component Class

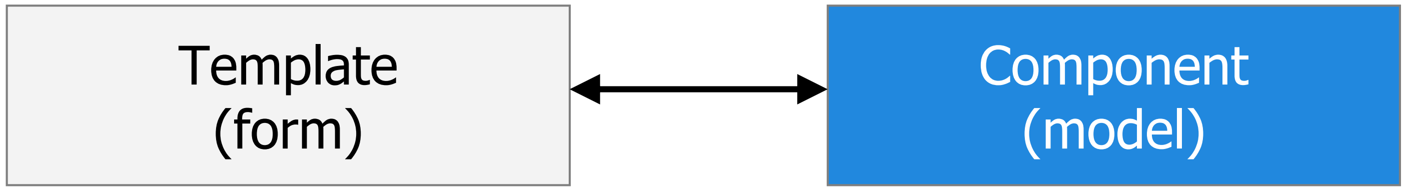
- **Data model**: properties for data binding
- **Methods** to handle events such as form submit

What are Template-Driven Forms?

- A component template is used to create a form and validate data provided by a model object (declarative approach)

  - Implicit creation of FormControl() by directives

- Two-way data binding, form control state and validation support are provided by using directives in the template



Importing FormsModule

- To get started using template-driven forms import **FormsModule**:

app.module.ts

```
import { NgModule }       from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule} from '@angular/forms';
import { AppComponent} from './app.component';

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

Import template-driven forms module

Using ngForm and ngModel

ngForm and **ngModel** directives work together to provide change state and validation functionality:

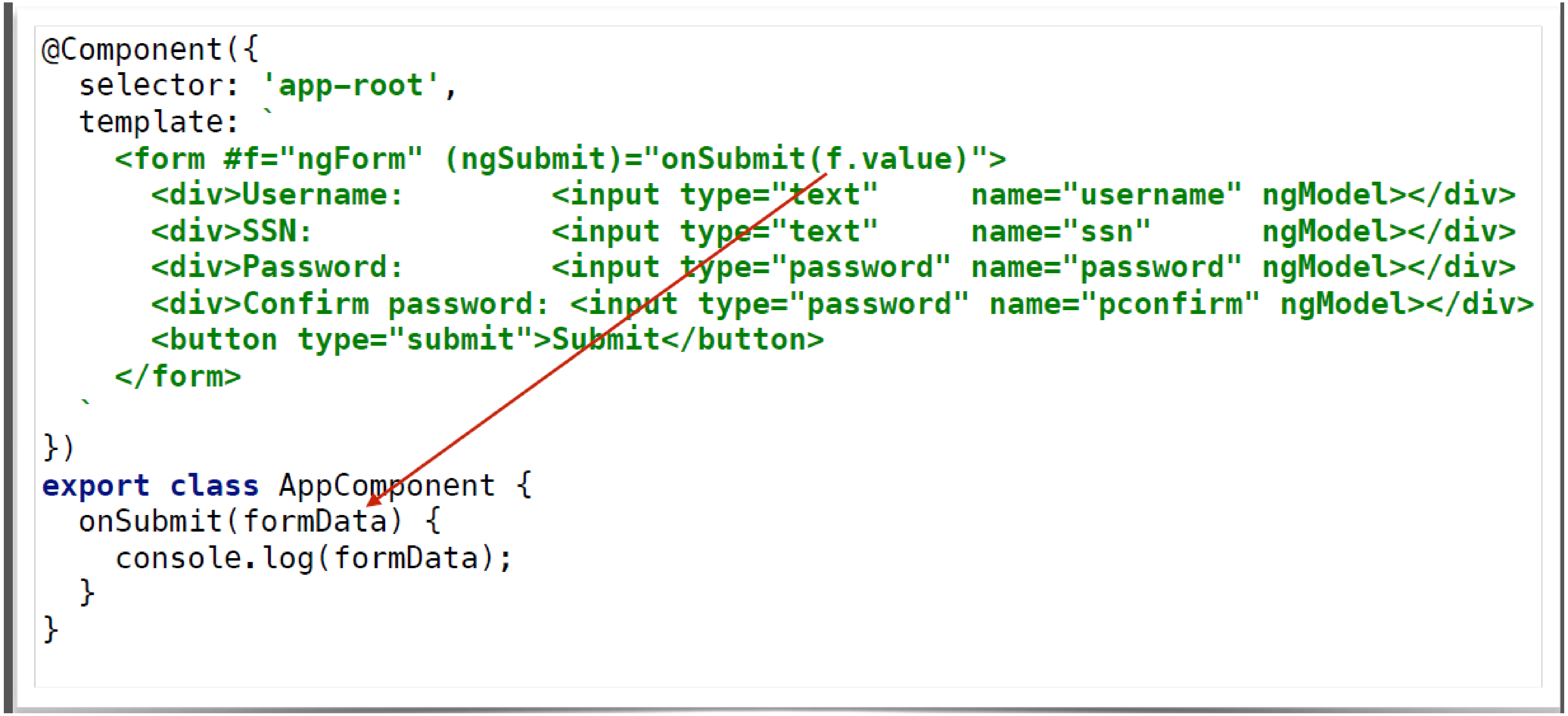
- Check if form/controls are dirty/pristine
- Check if form/controls are valid/invalid

Get to instance of form

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
    <input type="text" name="city" #city="ngModel" [(ngModel)]="city" />
    ...
</form>
```

Register control with ngForm instance

A template-driven form – Simple Example



```
@Component({
  selector: 'app-root',
  template: `
    <form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
      <div>Username:         <input type="text"     name="username" ngModel></div>
      <div>SSN:              <input type="text"     name="ssn"      ngModel></div>
      <div>Password:         <input type="password" name="password" ngModel></div>
      <div>Confirm password: <input type="password" name="pconfirm" ngModel></div>
      <button type="submit">Submit</button>
    </form>
  `
})
export class AppComponent {
  onSubmit(formData) {
    console.log(formData);
  }
}
```

Template-driven Directives

Form is mapped to FormGroup

FormGroup

```
<form #signupForm="ngForm" (ngSubmit)="save(signupForm)">
      <input type="text"
             id="firstName" name="firstName"
             [(ngModel)]=customer.firstName
             #firstName="ngModel" />
</form>
```

ngModel is used for two-way data binding to the data model

FormControl

Input element is mapped to FormControl

Define template reference variable to refer to the element anywhere in the template

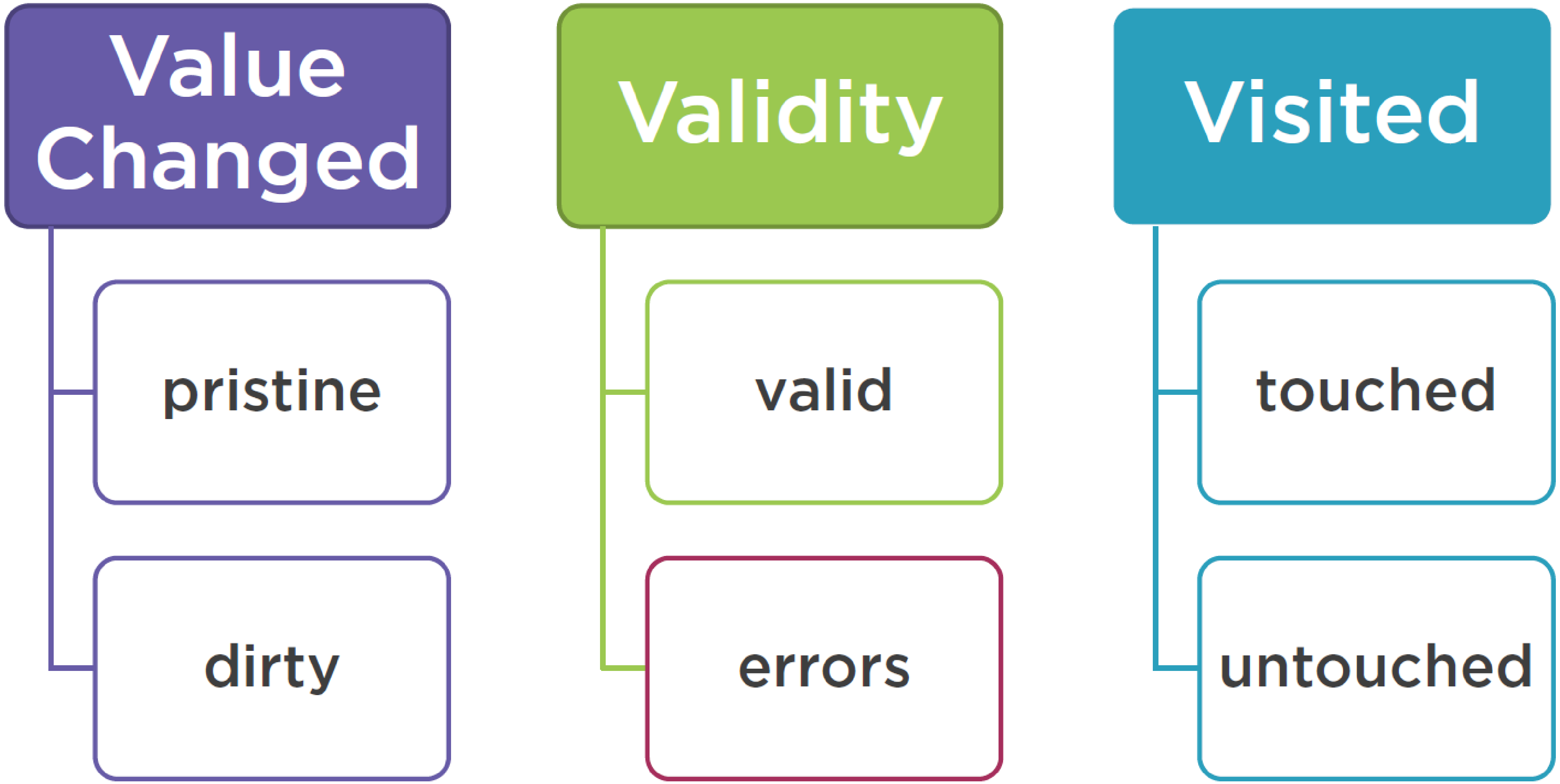
Show/Hide Validation Errors

- Use the **template reference variable** (e.g., #firstName) to access the state of the target control and determine if it's valid

form.component.html

```
<form #nameForm="ngForm" (ngSubmit)="onSubmit()">
  Name: <input type="text" name="firstName" [(ngModel)]="customer.firstName"
         #firstName="ngModel" required />

  <span [hidden]="firstName.valid || firstName.pristine">
     Name is invalid
  </span>
  <button type="submit" [disabled]="!nameForm.valid">Submit</submit>
</form>
```

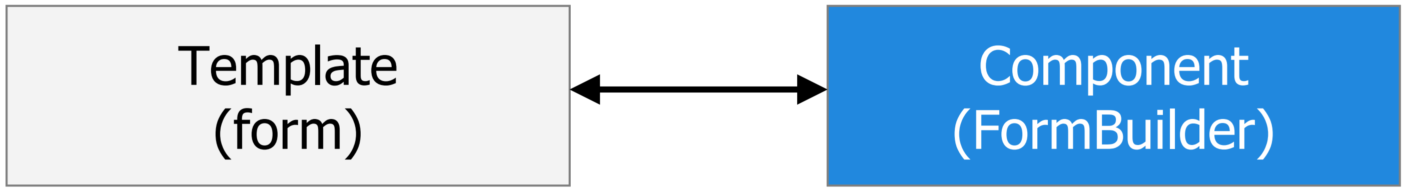
Form/Input Elements State



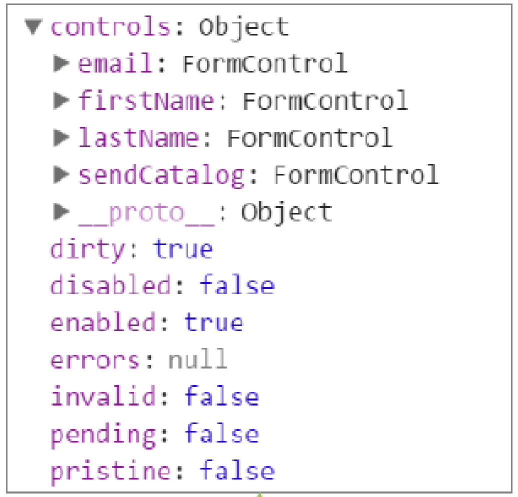
What are Reactive Forms?

- Component code defines controls and validators that are used in a form (imperative approach)

- Relies on the FormBuilder service to create controls and validators and organize them into one or more groups



Reactive Forms



```
▼ controls: Object
  ▶ email: FormControl
  ▶ firstName: FormControl
  ▶ lastName: FormControl
  ▶ sendCatalog: FormControl
  ▶ __proto__: Object
  dirty: true
  disabled: false
  enabled: true
  errors: null
  invalid: false
  pending: false
  pristine: false
```

Component Class:

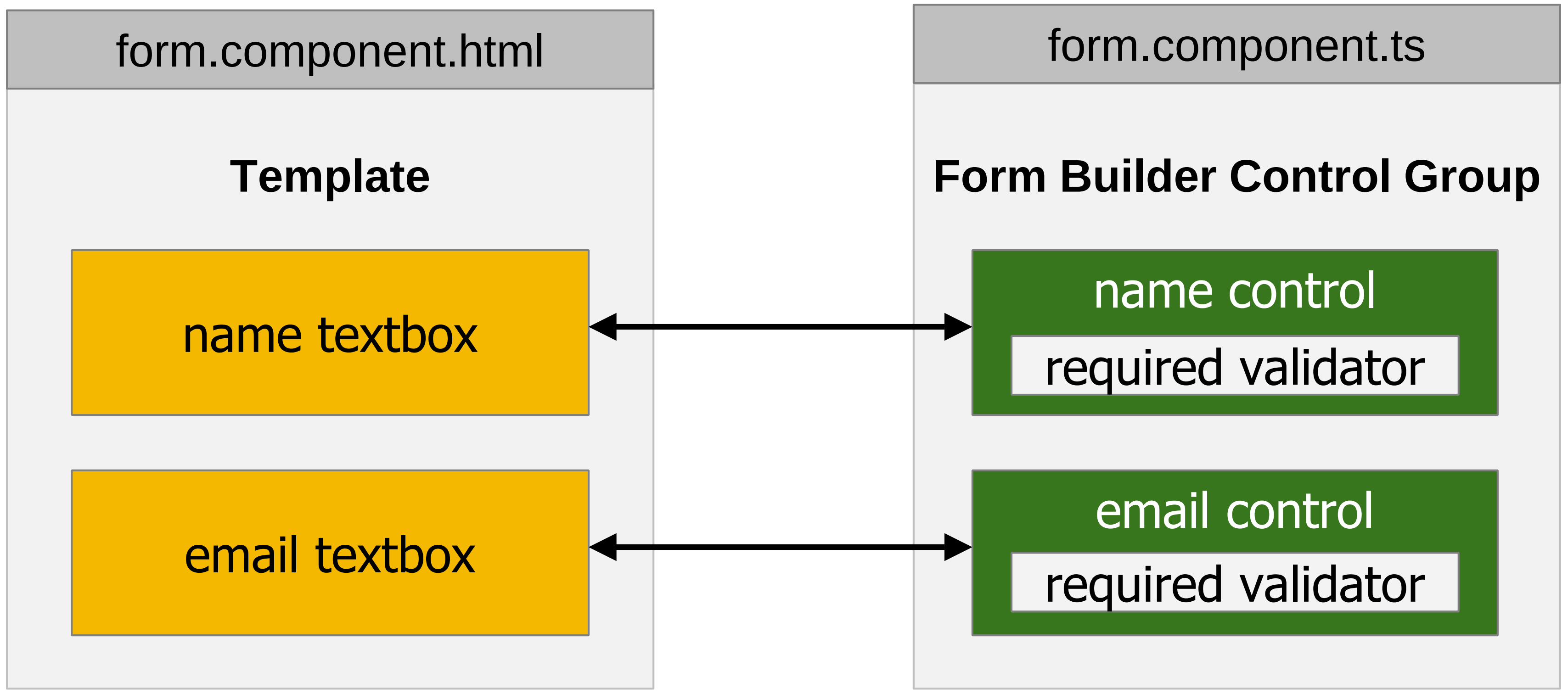
- Form model
- Validation rules
- Validation error messages
- Data Model: Properties for managing data
- Methods to handle events such as form submit

Template

- Form element
- Input element(s)
- Binding to form model

Reactive Forms Overview

- Form controls and validators are defined in component code
- Controls are bound to form input controls



Importing ReactiveFormsModule

- To get started using reactive forms import **ReactiveFormsModule**:

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule} from '@angular/forms';
import { AppComponent} from './app.component';

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

Import reactive forms module

Create a FormGroup using FormBuilder

- FormBuilder provides a group() function that can be used to create a control group

form.component.ts

```
@Component({  selector: 'model-driven-form' })
export class ModelFormComponent implements OnInit {
  form: FormGroup;
  constructor(private formBuilder: FormBuilder) { }
  ngOnInit() {
    this.model = new Hero(18, 'Dr IQ', 'Really Smart', 'Chuck Overstreet', 'iq@test.com');
    this.form = this.formBuilder.group({
      name:     [this.model.name, Validators.required],
      alterEgo: [this.model.alterEgo, Validators.required],
      email:    [this.model.email, [Validators.required, ValidationService.emailValidator]],
      power:    [this.model.power, Validators.required]
    });
  }
}
```

Add the formControlName directive

- Add formControlName to each form control to bind it to the respective "control" in the form group

form.component.html

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  Name:        <input type="text"  formControlName="name" />
  Alter Ego:   <input type="text"  formControlName="alterEgo" />
  Hero Email:  <input type="email" formControlName="email" />
  Power:       <select formControlName="power">
                 <option *ngFor="let p of powers" [value]="p">{{p}}</option>
               </select>
  ...

</form>
```

Show/Hide Validation Errors

- Use the FormGroup object to access controls and check validity

form.component.html

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  Name:         <input type="text" formControlName="name" />
                <div [hidden]="form.controls.name.valid">Name is required</div>

  Alter Ego:    <input type="text" formControlName="alterEgo" />
                <div [hidden]="form.controls.alterEgo.valid">
                   Alter Ego is required
                </div>
  ...

</form>
```

FormControl **tacks**:

Value

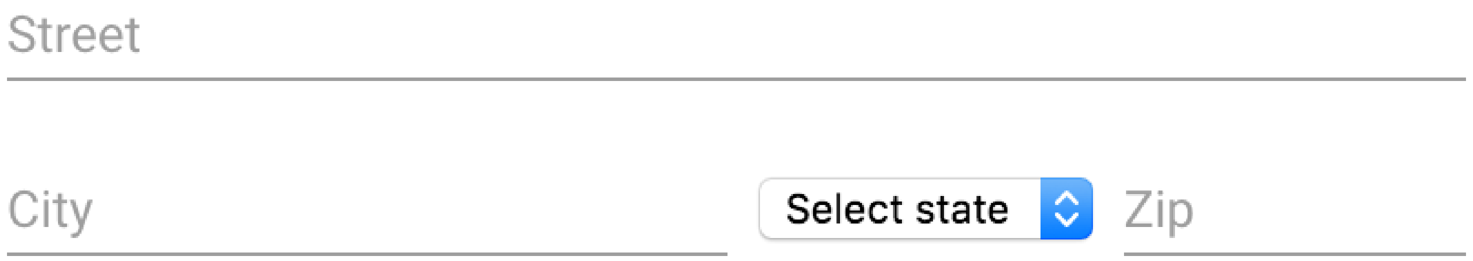
Validation status

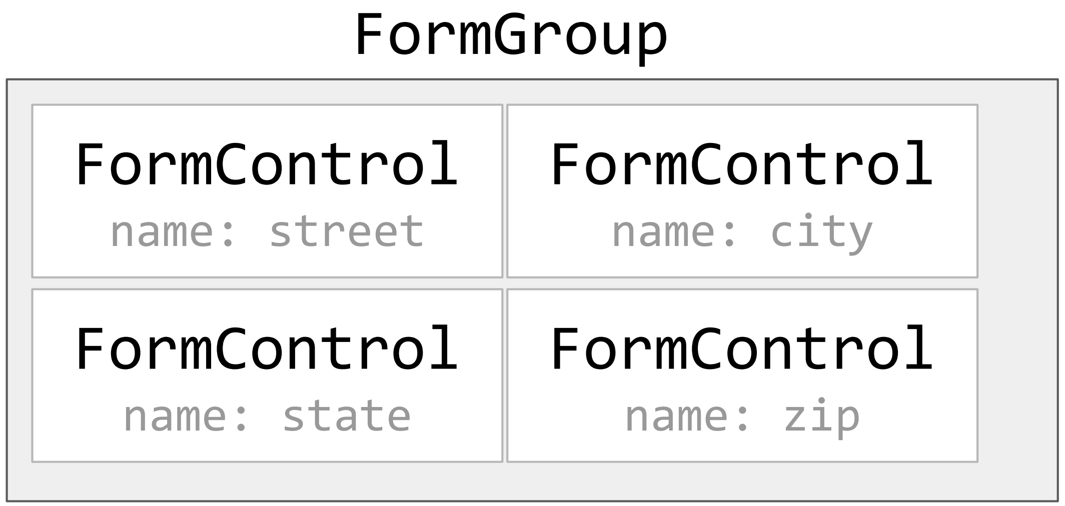
User interactions

Events

```
const control = new FormControl();

control.value                                           // null

control.status                                          // VALID

control.valid                                           // true

control.pristine                                        // true

control.untouched                                       // true
```

FormGroup Example

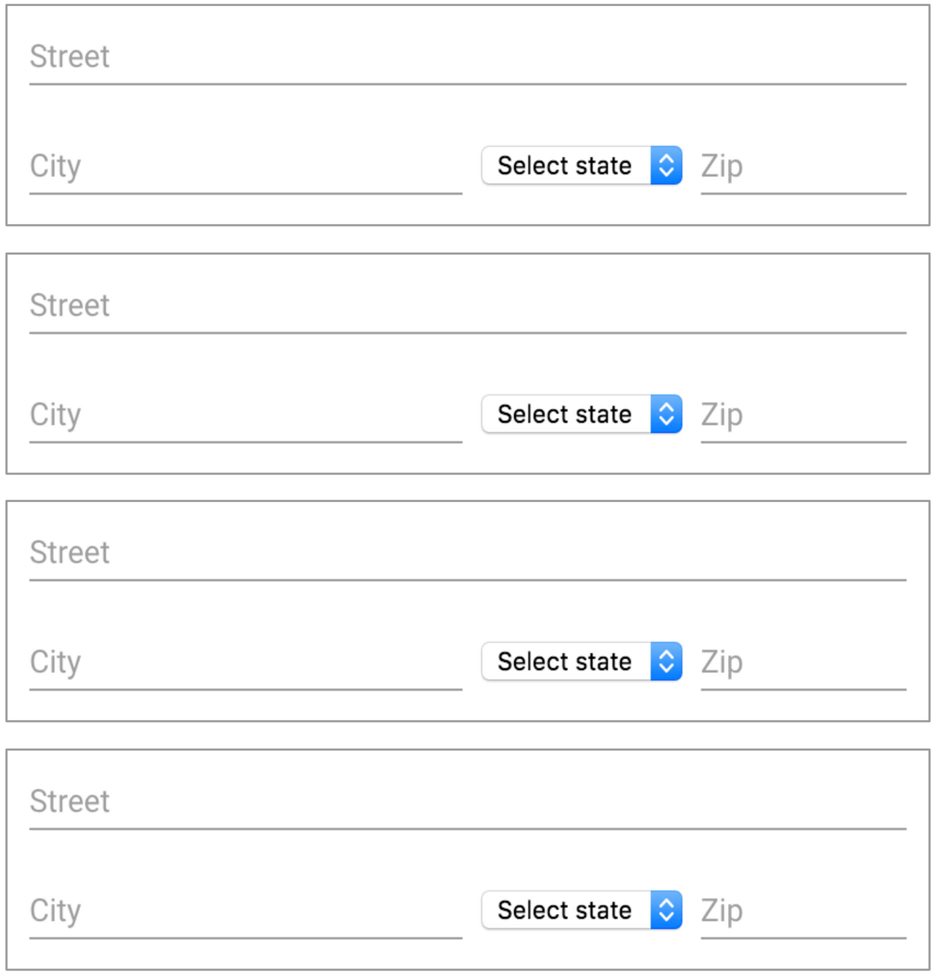


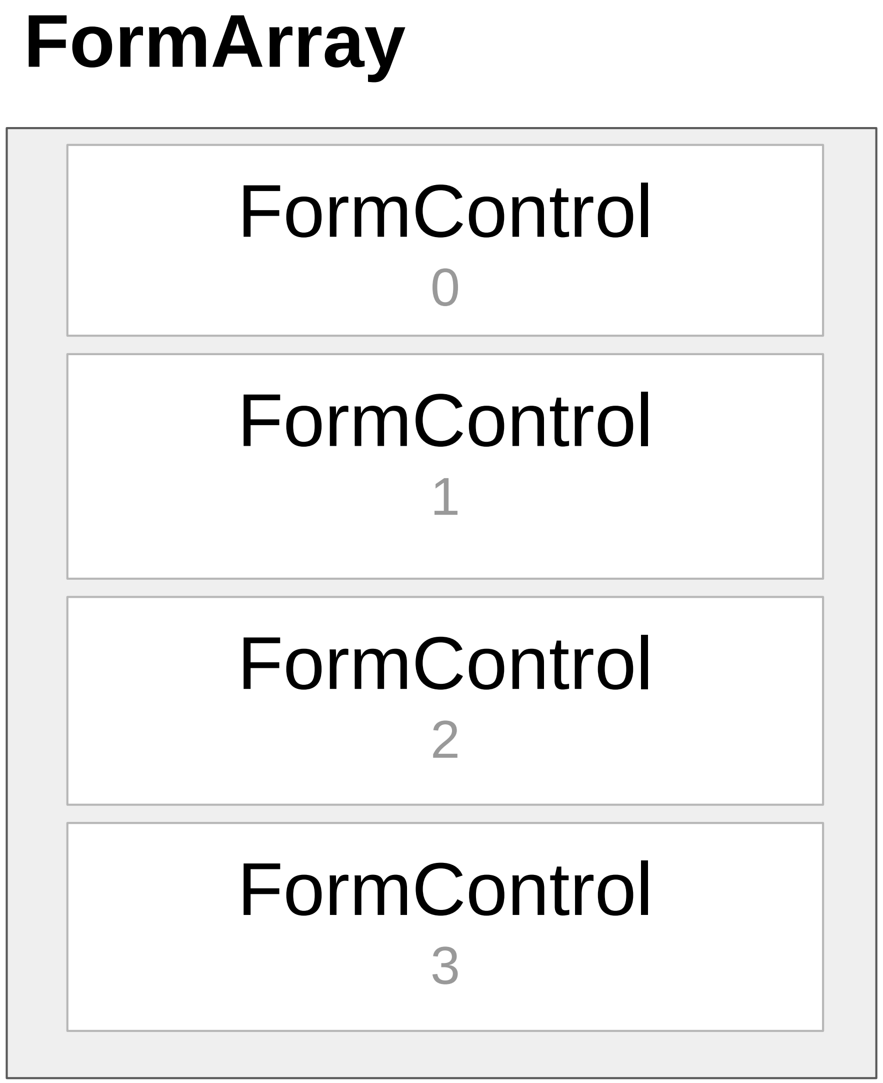


```
const form = new FormGroup({
  street: new FormControl('', Validators.required),
  city: new FormControl('')
});

form.value                                          // {street: '', city: ''}

form.status                                         // INVALID

form.setValue({street: '123 Majd St', city: ‘Doha'});

form.status                                         // VALID
```

FormArray

- Used for dynamically adding input elements to the form





CODE EXAMPLE

```
const arr = new FormArray([
  new FormControl('SF'),
  new FormControl('NY')
]);

arr.value                       // ['SF', 'NY']

arr.status                      // VALID

arr.setValue(['LA', 'LDN']);            // ['LA', 'LDN']

arr.push(new FormControl('MTV'));   // ['LA', 'LDN', 'MTV']
```

Reactive Form - Creating FormControls

customer.component.ts

```
...
import { FormGroup, FormControl } from '@angular/forms';


...
export class CustomerComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.customerForm = new FormGroup({
          firstName: new FormControl(),
          lastName: new FormControl(),
          email: new FormControl(),
          sendCatalog: new FormControl(true)});
  }
}
```

Reactive Form – Form Template

customer.component.html

```
<form (ngSubmit)="save()" [formGroup]="customerForm">

   <input type="text"  formControlName="firstName" />
   <input type="text"  formControlName="lastName" />
   <input type="email" formControlName="email" />
   ...
</form>
```

Bind the form element to the **FormGroup** property

Bind each input element to its associated **FormControl**

Accessing the Form Model Properties

```
customerForm.get('firstName').valid
```

Or

```
customerForm.controls.firstName.valid
```

Using setValue and patchValue

- Use **setValue** to initialize all form controls and use **patchValue** to initialize some controls of the form

```
this.customerForm.setValue({
    firstName: 'Ali',
    lastName: 'Mujtahid',
    email: 'ali@test.com'
});

this.customerForm.patchValue({
    firstName: 'Ali',
    lastName: 'Mujtahid',
});
```

Using FormBuilder to simply the Form Creation

- Import FormBuilder
- Inject the FormBuilder instance
- Use that instance

```
import { FormBuilder } from '@angular/forms';
...
export class CustomerComponent  {
        ...
        constructor(private fb: FormBuilder) { }
        this.customerForm = this.fb.group({
            firstName: '',
            lastName: '',
            email: '',
            sendCatalog: true
        });
    }
 }
```

Setting Built-in Validation Rules

- Pass in the validator or array of validators when creating a form control

```
this.customerForm = this.fb.group({
    firstName: ['', [Validators.required, Validators.minLength(3)]],
    sendCatalog: true
});
```

Adjusting Validation Rules at Runtime

- Determine when to change validation (e.g., make phone required if the customer selects notification by SMS)
- Use setValidators or clearValidators
- Call updateValueAndValidity

```
setNotification(notifyVia: string) {
    const p = this. customerForm.get('phone');
    if (notifyVia === 'text') {
        p.setValidators(Validators.required);
    } else {
        p.clearValidators();
    }
    p.updateValueAndValidity();
}
```

Watching Form ValueChanges

- Use the valueChanges Observable property
- Subscribe to the Observable

```
this.customerForm.valueChanges.subscribe(value =>
    console.log(JSON.stringify(value)));
```

- Then react such as enforcing custom Validation rules, and provide automatic suggestions

Services

Services

A Service provides anything our application needs.
It often shares data or functions between modules

Why Build a Data Access Service?

- Separation of Concerns
- Reusability
- Data Sharing

Saving Edits

```
let product = Object.assign({}, this.product, this.productForm.value);
```

Service

Provides something of value

Shared data or logic

e.g. Data, logger, exception handler, or message service

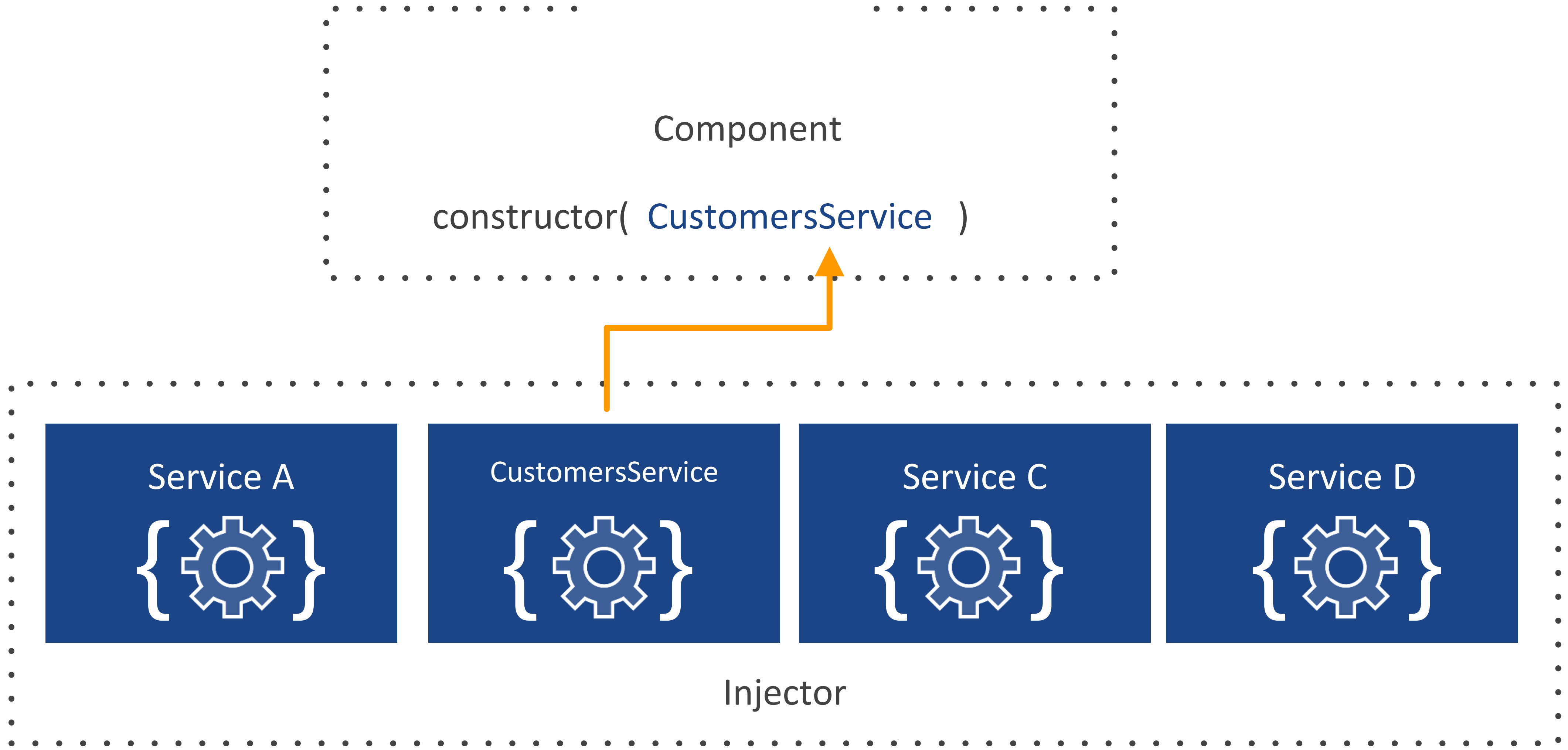
customers.service.ts

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

@Injectable()
export class CustomersService {

    getCustomers() {
        return …;
    }

}
```

Service is just a class

Dependency Injection

Dependency Injection is how we provide an instance of a class to another Angular component


Component
constructor(CustomersService)
Service A
CustomersService
Service C
Service D
Injector

Registering a Service Provider

Services must have a provider in order to be injected

app.module.ts

```
import { NgModule } from '@angular/core';
import { CustomersService } from
'../core/services/customers.service';

@NgModule({
  …
  providers: [ CustomersService ]
})
export class AppModule { }
```

Create a provider for service

Injecting a Service into a Component

Locates the service in the Angular injector

Injects into the constructor

customers.component.ts

```
export class CustomersComponent implements OnInit {
    customers: ICustomer[];

    constructor(private customersService: CustomersService) { }

    ngOnInit() {
       this.customers = this.customersService.getCustomers();
    }
}
```

Inject service

Injecting a Service into a Service

Same concept as injecting into a Component

customers.service.ts

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

  getCustomers() {
    return this.http.get<ICustomer[]>(customersUrl);
  }
}
```

Provides metadata about injectables

Injecting HttpClient

HttpClient

HttpClient

We use HttpClient to get and save data with Promises or Observables. We isolate the http calls in a shared Service.

HttpClient Step by Step

Import the HttpClientModule

Inject HttpClient in a service

Call http.get()

Subscribe to the Service's function in the Component

Http Requirements

HttpClientModule contains the providers for Http

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule  } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';
import { CustomersComponent } from './customers.component';


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

Import HttpClientModule

Using Http

customers.service.ts

Inject

Perform Http GET

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

  getCustomers() : Observable<ICustomer[]> {
    return this.http
      .get<ICustomer[]>('api/customers);
  }

}
```

Get the response

Subscribing to the Observable

Component is handed an Observable

We Subscribe to it

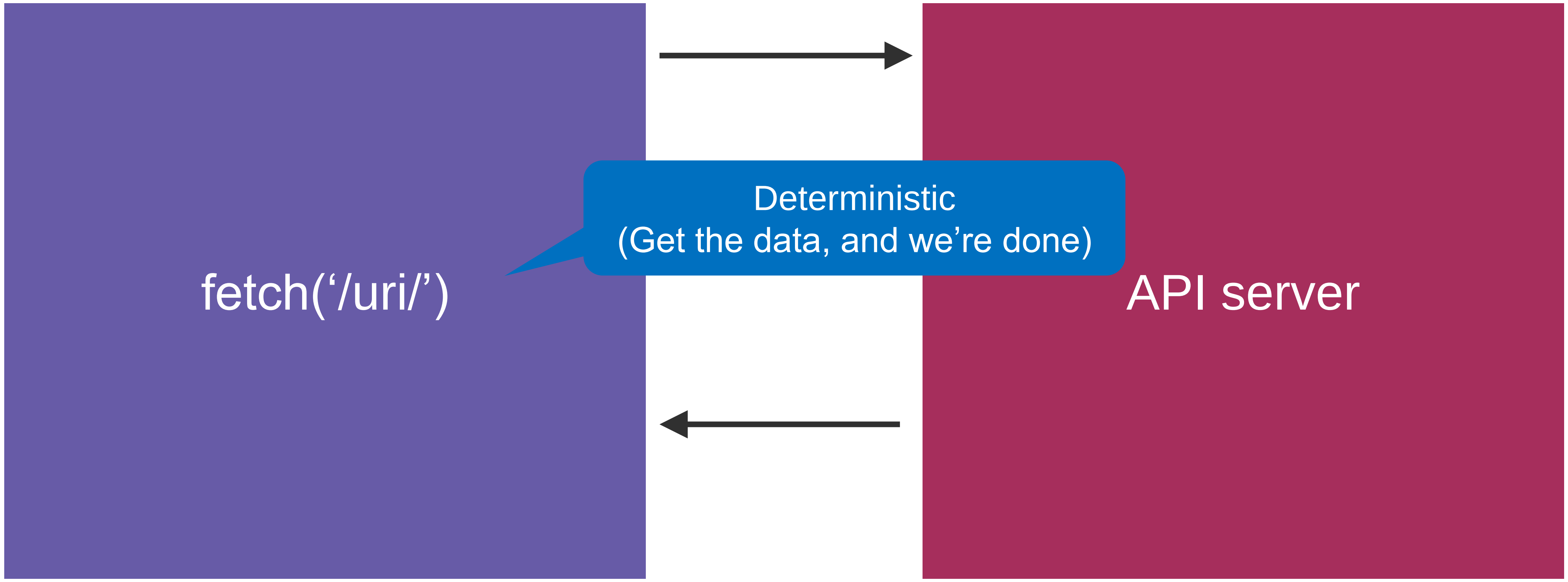
customers.component.ts

```
constructor(private customersService: CustomersService) { }

getCustomers() {
  this.customers = [];

  this.customersService.getCustomers()
    .subscribe(
      customers => this.customers = customers,
      error => this.errorMessage = error
    );
}
```

Subscribe to the Observable

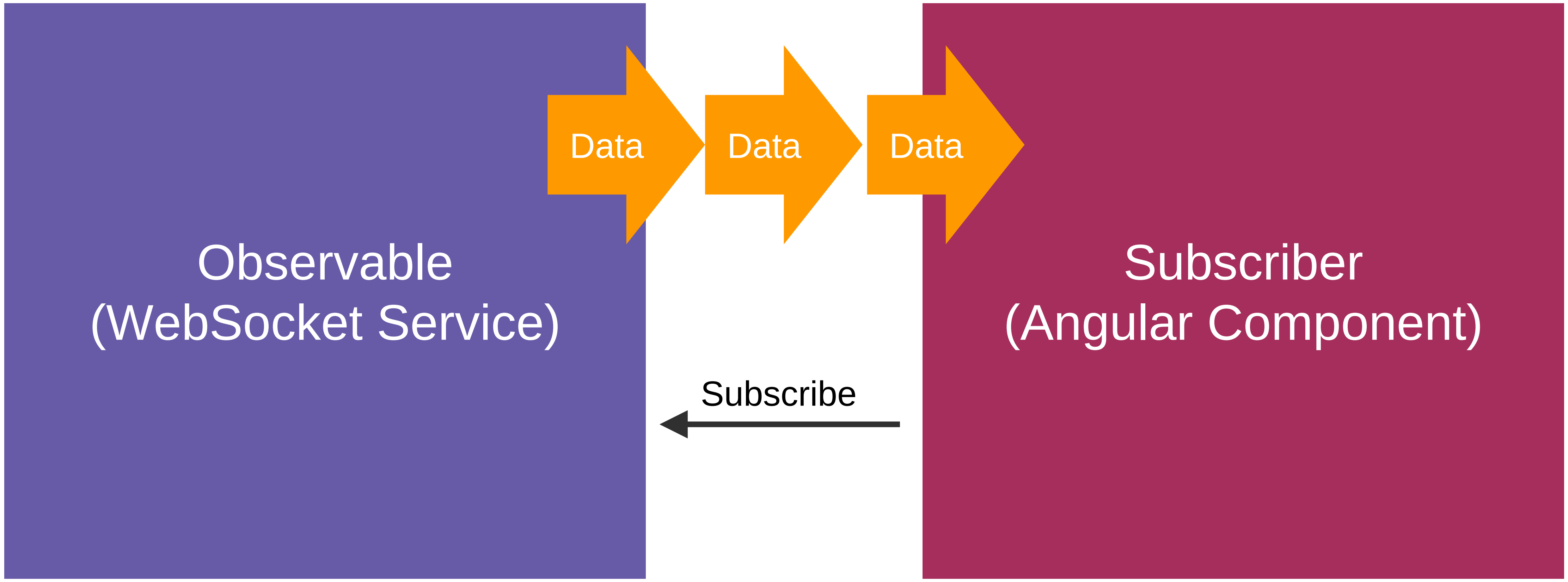
Handle error conditions

Promise Overview



RxJS Observables Overview

Observable is a collection of events that arrive asynchronously over time



http://reactivex.io

Routing

Angular Routing

- Components can be changed/swapped by using routing
- Import and use RouterModule from **@angular/router**
- Can define parent and child routes

Import Routes and RouterModule

RouterModule gives us access to routing features

Routes help us declare or route definitions

app-routing.module.ts

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

Import routing features

Defining Routes

Define the route’s path

Indicate parameters with :

Set the component that we’ll route to

app-routing.module.ts

When I see this path, go to this component

```
const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'customers' },
  { path: 'customers', component: CustomersComponent },
  { path: 'customers/:id', component: CustomerComponent },
  { path: '**', pathMatch: 'full', component:
PageNotFoundComponent },
];
```

Define a Module

Create a routing module using our routes, and import it

Export our new AppRoutingModule

app-routing.module.ts

Only use forRoot() for the app root module’s routes

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

Routing, All Together

app-routing.module.ts

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

const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'customers', },
  { path: 'customers', component: CustomersComponent },
  { path: 'customers/:id', component: CustomerComponent },
  { path: '**', pathMatch: 'full', component: PageNotFoundComponent },
];

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

```
@Component({
  selector: 'app-container',
  template: `<router-outlet></router-outlet>`
})
export class AppComponent { }
```

Define where components get loaded in the application

app.component.html

RouterOutlet

Angular puts components in a "component container"

<router-outlet> defines location where components are loaded

RouterLink Directive

The routerLink directive can be used to add links to routes
Defines the route path and any route parameter data

customer.component.ts

```
@Component({
  selector: 'customers',
  templateUrl: './customers.component.html'
})
export class CustomersComponent {
  // …
}
```

customer.component.html

```
<a routerLink="/customers">
  Customers
</a>

<a [routerLink]="['/customers', customer.id]">
  {{ customer.firstName }}
</a>
```

Reading Parameters from a Route

```
{ path: 'productEdit/:id', component: ProductEditComponent }
```

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

```
constructor(private route: ActivatedRoute) {
   let id = +this.route.snapshot.params['id'];
   ...
}
```

Or

```
constructor(private route: ActivatedRoute) {
   this.sub = this.route.params.subscribe(
      params => {
         let id = +params['id'];
         ...
      }
   );
}
```

Route Parameters

Snapshot

Easiest, as long as parameter values do not change

Observable

Gets new values as parameters change when component is re-used

Snapshot Parameters

customer.component.ts

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

export class CustomerComponent implements OnInit {
  private id: any;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
      this.id = parseInt(this.route.snapshot.params['id'], 10);
      this.getCustomer();
    }
  }

  // ...
}
```

Access route information

Grab the snapshot of the current route parameters

Observables and Parameters

customer.component.ts

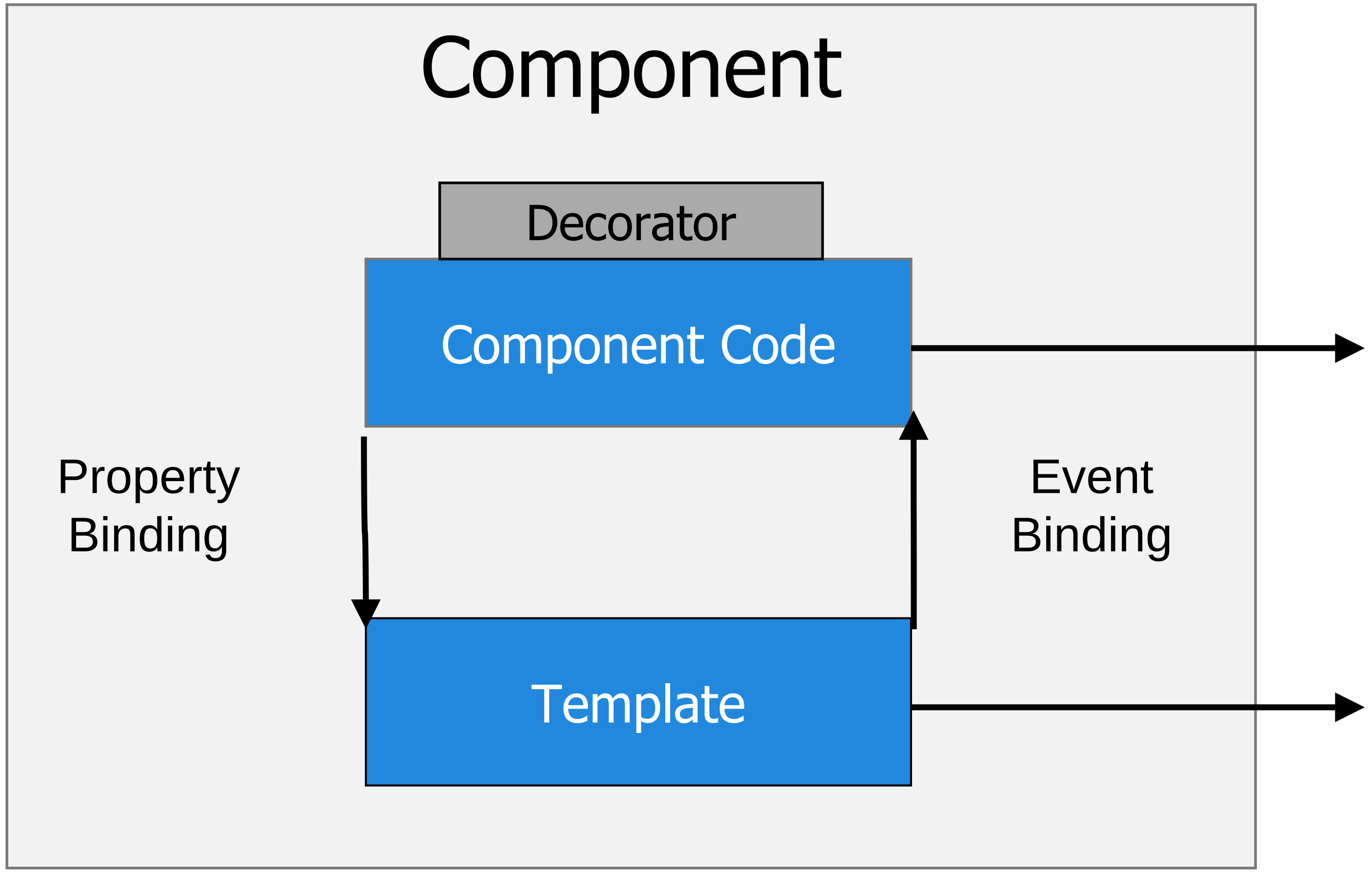
```
import { ActivatedRoute } from '@angular/router';

export class CustomerComponent implements OnInit {
  private id: any;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
      this.route.params
        .map(params => params['id'])
        .do(id => this.id = +id)
        .subscribe(id => this.getCustomer());
    }
  }
  // ...
}
```

Access route information

Get route parameters, as they change. Ideal when routing to the same component.

ngModule

Component Code and Templates



```
@Component({
   templateUrl: './mytemplate.html'
})
export class MyComponent {
   isHidden = true;
   clickMe() { … }
}
```

```
<div [hidden]="isHidden">
    <span (click)="clickMe()">
      Section 3
    </span>
</div>
```

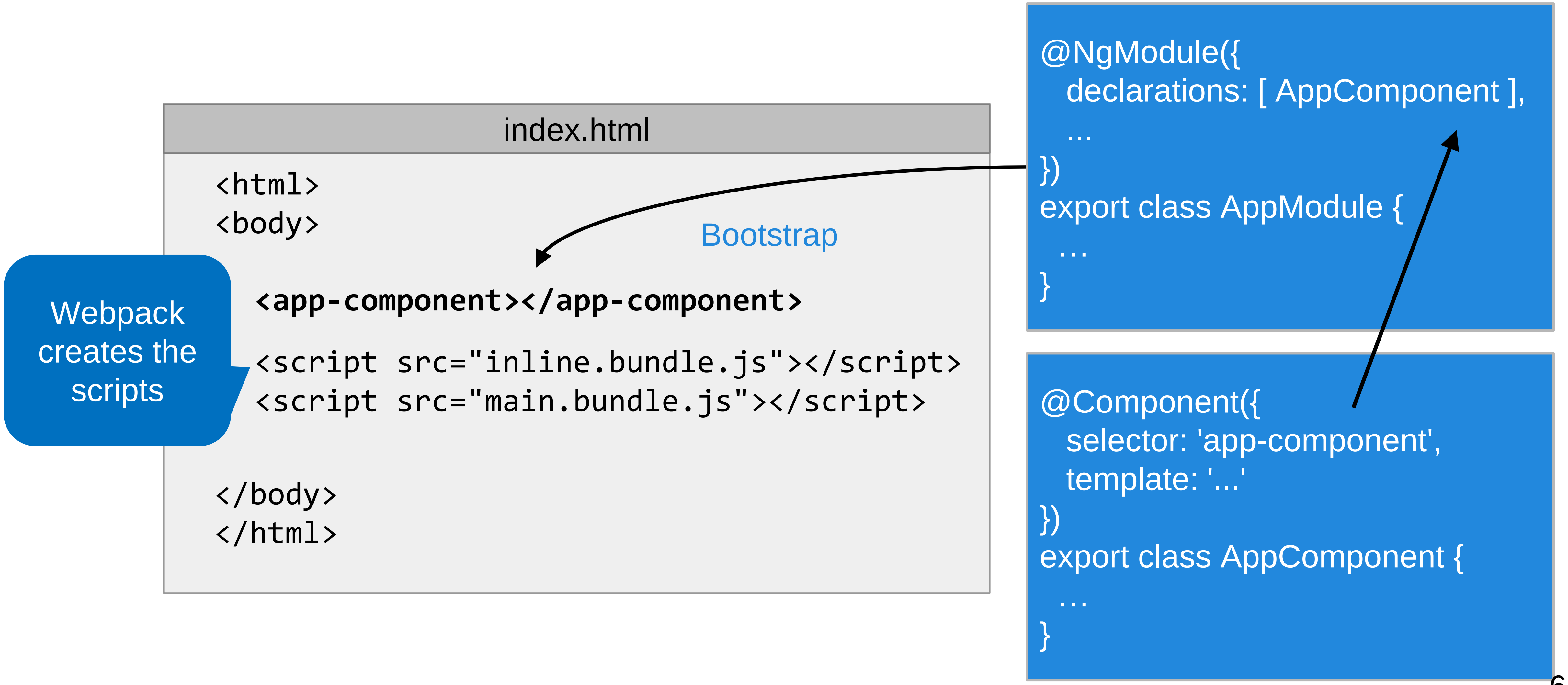
NgModules

NgModules help organize an application

app.module.ts

```
import { NgModule }           from '@angular/core';
import { BrowserModule }      from '@angular/platform-browser';
import { FormsModule }        from '@angular/forms';
import { HttpClientModule }  from '@angular/common/http';

import { AppComponent }  from './app.component';

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

Components, Modules and Bootstrapping



```
<html>
<body>

  <app-component></app-component>

  <script src="inline.bundle.js"></script>
  <script src="main.bundle.js"></script>

</body>
</html>
```

```
@NgModule({
  declarations: [ AppComponent ],
  ...
})
export class AppModule {
  …
}
```

```
@Component({
  selector: 'app-component',
  template: '...'
})
export class AppComponent {
  …
}
```

Bootstrapping Angular

Applications must bootstrap a root app module

Import the **platformBrowserDynamic**() function and pass the root app module

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule }              from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .then(success => console.log(`Bootstrap success`))
  .catch(err => console.error(err));
```

Angular CLI

Key Angular CLI Commands

```
ng --help

ng new [my-app-name]

ng [g | generate] [component | directive | route | pipe |
service ]

ng build

ng serve

ng serve –o   --run and watch for changes
```