Angular Reactive Forms

Angular Forms

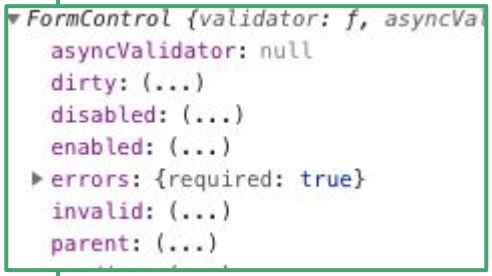
Template-driven:

- Easy to use
- Similar to Angular 1
- Two-way data binding -> Minimal component code
- Automatically tracks form and input element state

Reactive:

- More flexible -> more complex scenarios
- Immutable data model
- Easier to perform an action on a value change
- Reactive transformations -> DebounceTime or DistinctUntilChanged
- Easily add input elements dynamically
- Easier unit testing

Template-driven



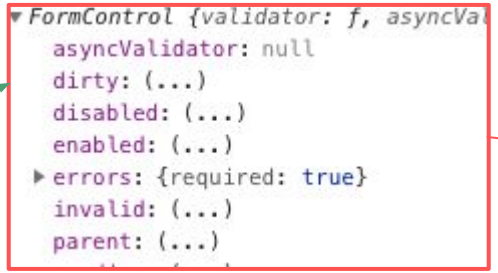
Template

public firstName = ‘Ethan’,
public lastName = ‘Hunt’,
public email = ‘’,
public sendCatalog = false,

Component

Reactive

Template



Component

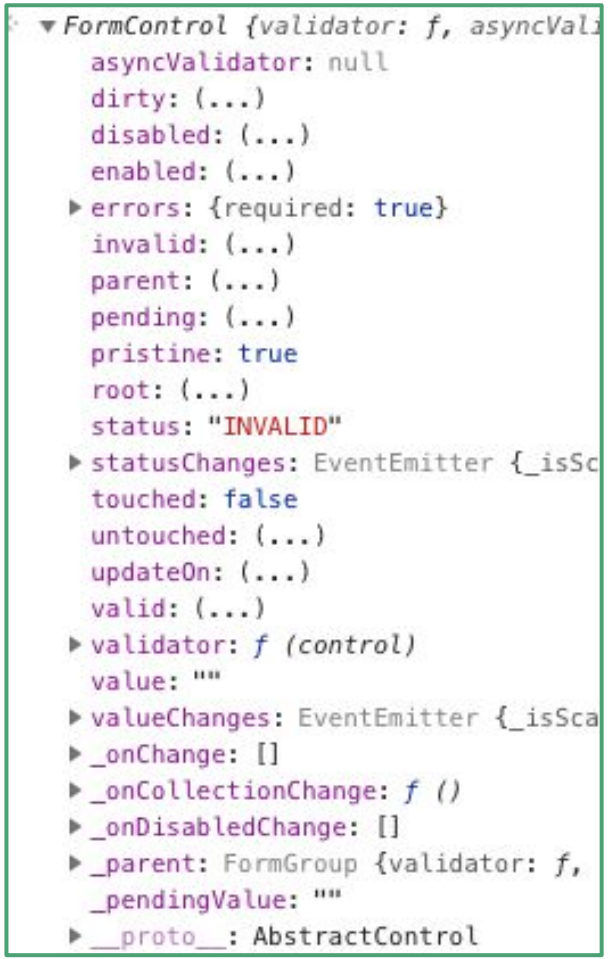
public firstName = ‘Ethan’,
public lastName = ‘Hunt’,
public email = ‘’,
public sendCatalog = false,

Building Reactive Forms

- The Component Class
- The Angular Module
- The Template
- Using setValue and patchValue
- Simplifying with FormBuilder

Form Model

- Root FormGroup
- FormControl for each input element
- Nested FormGroups as desired
- FormArrays



Creating a FormGroup

customer.component.ts

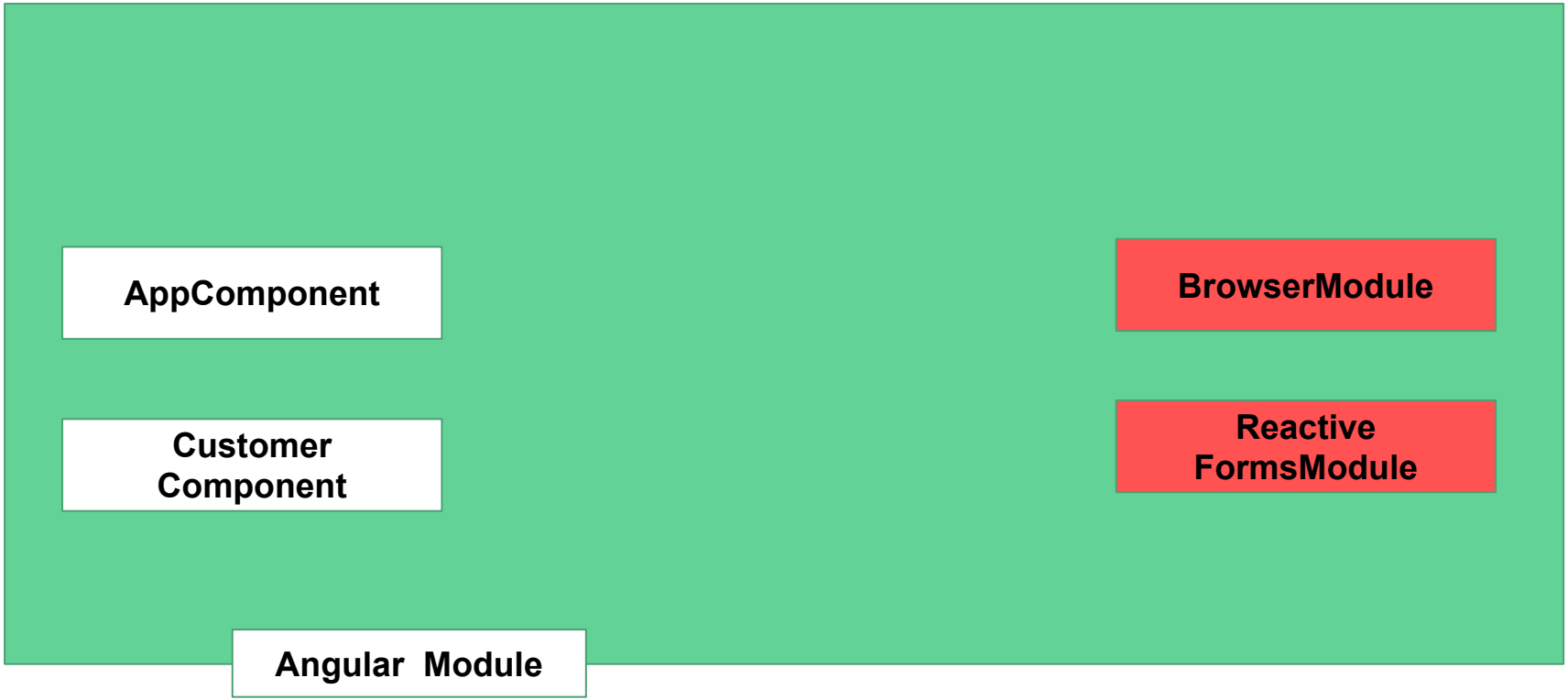
```
…
Import { FormGroup } from ‘@angular/forms’;
…
export class CustomerComponent implements OnInit {
        customerForm: FormGroup;
        customer: Customer = new Customer();

        ngOnit(): void {
                this.customerForm = new FormGroup( { });
        }
}
```

Creating a FormControls

customer.component.ts

```
…
Import { FormGroup, FormControl } from ‘@angular/forms’;
…
export class CustomerComponent implements OnInit {
        ...
        ngOnit(): void {
                this.customerForm = new FormGroup( {
                        firstName : new FormControl(),
                        lastName : new FormControl(),
                        sendCatalog: new FormControl(true)});
        }
}
```


AppComponent
Customer Component
BrowserModule
Reactive FormsModule
Angular Module

Reactive Forms Directives

Reactive Forms

- **formGroup**
- **formControl**
- **formControlName**
- **formGroupName**
- **formArrayName**

formGroup

customer.component.html

```
<form (ngSubmit)=”save()” [formGroup]=”customerForm”>
      …
</form>
```

formControlName

customer.component.html

```
<form (ngSubmit)=”save()” [formGroup]=”customerForm”>
    <fieldset>
        <label for=”firstNameId”>First Name</label>
        <input id=”firstNameId” type=”text”
                formControlName=”firstName”>
    </fieldset>
</form>
```

Accessing the Form Model Properties

```
customerForm.controls.firstName.valid

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

```
firstName = new FormControl();

ngOnInit(): void {
    this.customerForm = new FormGroup({
        firstName: this.firstName,
        ...
    });
}
```

Using setValue and patchValue

```
this.customerForm.setValue({
        firstName: ‘Jack’,
        lastName: ‘Harkness’,
        Email: ‘jack@torchwood.com’
});
```

```
this.customerForm.patchValue({
        firstName: ‘Jack’,
        lastName: ‘Harkness’
});
```

FormBuilder

- Creates a form model from a configuration
- Shortens boilerplate code
- Provided as a service

FormBuilder steps

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

```
Import {
        FormBuilder
} from '@angular/forms';
```

```
constructor(
        Private fb: FormBuilder
) {}
```

```
this.customerForm.setValue({
        firstName: null,
        lastName: null,
        Email: null
});
```

FormBuilder's FormControl Syntax

```
this.customerForm = this.fb.group({
        firstName: '',
        sendCatalog: true
});
```

```
this.customerForm = this.fb.group({
        firstName: {value: 'n/a', disabled: true},
        sendCatalog: { value: true, disabled: false}
});
```

```
this.customerForm = this.fb.group({
        firstName: [''],
        sendCatalog: [{ value: true, disabled: false}]
});
```

Validation

- Setting Built-in Validation Rules
- Adjusting Validation Rules at Runtime
- Custom Validators
- Custom Validators with Parameters
- Cross-field Validation

Setting Built-in Validation Rules

```
this.customerForm = this.fb.group({
            firstName: ['', [Validators.required,
Validators.minLength(3)]],
            lastName: ['', [Validators.required,
Validators.maxLength(50)]]
});
```

Adjusting Validation Rules at Runtime

```
phoneControl.setValidators(Validators.required);

phoneControl.clearValidators();

phoneControl.updateValueAndValidity();
```

Custom Validators

```
function myCustomValidator(c: AbstractControl): {[key: string]: boolean} |
null  {
        if (somethingIsWrong) {
            return { 'myvalidator': true };
        };
        return null;
}
```

Custom Validators with Parameters

```
function myCustomValidator(param: any): ValidatorFn {
(c: AbstractControl): {[key: string]: boolean} | null => {
        if (somethingIsWrong) {
            return { 'myvalidator': true };
        };
        return null;

}
```

Cross-field Validation

```
this.customerForm = this.fb.group({
      emailGroup: this.fb.group({
         email: ['', [Validators.required,
Validators.pattern('[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+')]],
         confirmEmail: ['', Validators.required],
            }, {validator: emailMatcher})
)};
```

Reacting to changes

- Watching
- Reacting
- Reactive Transformation

Watching

- valueChanges property emits events on value changes
- valueChanges is an Observable<any>
- Observable is a collection of events that arrive asynchronously over time
- Subscribe to the observable to watch the events
- statusChanges property emits events on validation changes

Reacting

- Validation rules
- Validation messages
- User interface elements
- Automatic suggestions
- And more...

Reactive Transformation

debounceTime

- Ignores events until a specific time has passed without another event
- debounceTime(1000) waits for 1000 milliseconds (1 sec) of no events before emitting another event

throttleTime

Emits a value, then ignores subsequent values for a specific amount of time

distinctUntilChanged

Suppresses duplicate consecutive items

Dynamically Duplicate Input Elements

- Define the input element(s) to duplicate
- Define a FormGroup, if needed
- Refactor to make copies
- Create a FormArray
- Loop through the FormArray
- Duplicate the input element(s)

Reactive Form in Context

- Sample Application
- Routing to the Form
- Reading a Route Parameter
- Setting a canDeactivate Guard
- Refactoring to a Custom Validation Class

CRUD Using HTTP

- Data Access Service
- Creating Data
- Reading Data
- Updating Data
- Deleting Data

Sending an HTTP Request