Angular Master Class

Forms

Goals

Forms in single page apps usually need to fulfill the following goals:

Goals

Forms in single page apps usually need to fulfill the following goals:

- Prevent the form from causing a page reload when submitting

Goals

Forms in single page apps usually need to fulfill the following goals:

- Prevent the form from causing a page reload when submitting
- Validate form controls before form is submitted

Goals

Forms in single page apps usually need to fulfill the following goals:

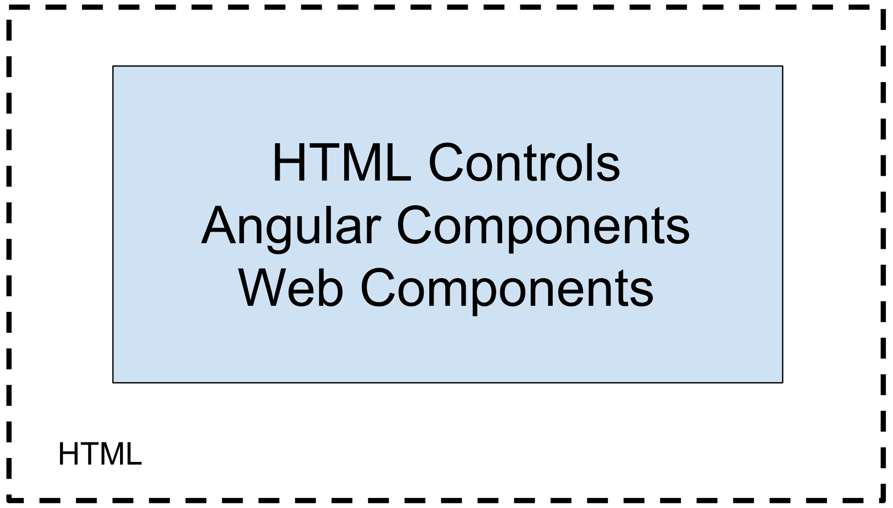
- Prevent the form from causing a page reload when submitting
- Validate form controls before form is submitted
- Display error and validation methods as user interacts with form

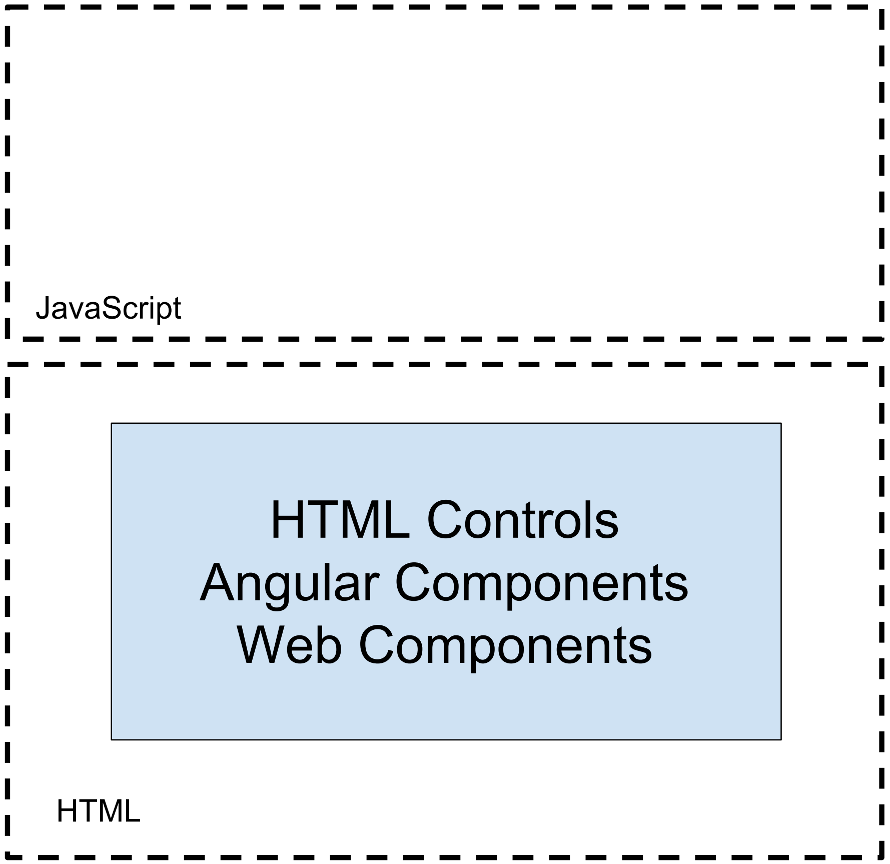
Forms in Angular

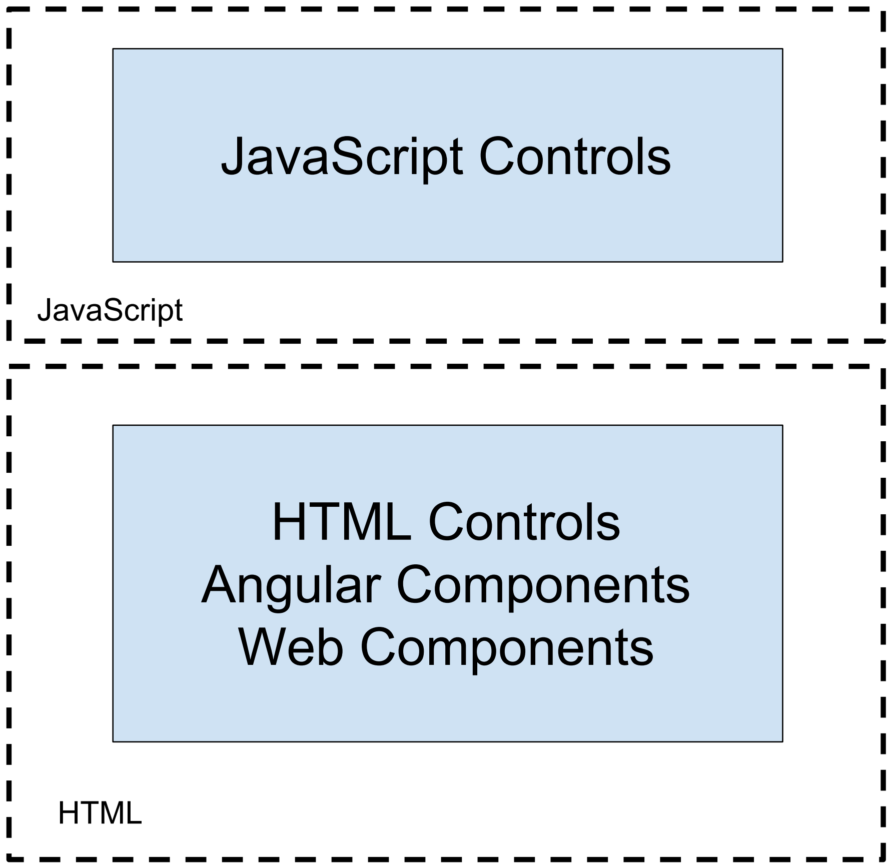
Angular provides two different ways of creating forms:

- **Template-driven** - Everything we need to build forms declaratively
- **Reactive** - Enables testing forms, reactive data-flow and thinner markup

HTML


HTML Controls
Angular Components
Web Components
HTML


JavaScript
HTML Controls
Angular Components
Web Components
HTML


JavaScript Controls
JavaScript
HTML Controls
Angular Components
Web Components
HTML

Template-driven Forms

Activate Form API

We activate the new Form API by importing **FormsModule**

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

@NgModule({
  imports: [
    ...
    FormsModule
  ]
  ...
})
class ContactsModule {}
```

Activate Form API

We activate the new Form API by importing **FormsModule**

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

@NgModule({
  imports: [
    ...
    FormsModule
  ]
  ...
})
class ContactsModule {}
```

ngForm Directive

`ngForm` provides us information about the current state of the form including:

ngForm Directive

`ngForm` provides us information about the current state of the form including:

- A JSON representation of the form value

ngForm Directive

`ngForm` provides us information about the current state of the form including:

- A JSON representation of the form value
- Validity state of the entire form

ngForm Directive

`ngForm` provides us information about the current state of the form including:

- A JSON representation of the form value
- Validity state of the entire form
- Also delegates submission events

```
<form>
  <label>Firstname:</label>
  <input type="text">

  <label>Lastname:</label>
  <input type="text">

  <button type="submit">Save</button>
</form>
```

```
<form #form="ngForm">
  <label>Firstname:</label>
  <input type="text">

  <label>Lastname:</label>
  <input type="text">

  <button type="submit">Save</button>
</form>
```

Accessing `ngForm` instance with local template variable.

Submitting Forms

We bind to the `ngSubmit` event to handle form submissions.

Submitting Forms

We bind to the `ngSubmit` event to handle form submissions.

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

  ...

</form>
```

Doesn't cause page reload when error is thrown.

Accessing Form values

`ngForm.value` returns a JSON representation of the form's control values.

```
<form
  #form="ngForm"
  (ngSubmit)="submit(form.value)">

  ...

</form>
```

Demo →

Adding Form Controls

We register form controls at a form using `ngModel` directive `name` attribute.

```
<form>

  <label>Firstname:</label>
  <input>
  ...

</form>
```

Adding Form Controls

We register form controls at a form using `ngModel` directive `name` attribute.

```
<form>

  <label>Firstname:</label>
  <input name="firstname" ngModel>
  ...

</form>
```

Control structure will be represented in `ngForm.value`

Adding Control Groups

Sometimes, controls need to be semantically grouped. `ngModelGroup` enables us to do that.

```
<div ngModelGroup="name">

  <label>Firstname:</label>
  <input name="firstname" ngModel>

  <label>Lastname:</label>
  <input name="lastname" ngModel>

</div>
```

Adding Control Groups

Sometimes, controls need to be semantically grouped. `ngModelGroup` enables us to do that.

```
<div ngModelGroup="name">

  <label>Firstname:</label>
  <input name="firstname" ngModel>

  <label>Lastname:</label>
  <input name="lastname" ngModel>

</div>
```

Demo →

Exercise: Template-driven form

Built-in Validation and Messages

Built-in Validation

Angular comes with the following built-in validation directives:

Built-in Validation

Angular comes with the following built-in validation directives:

- `required` - Value mustn't be empty

Built-in Validation

Angular comes with the following built-in validation directives:

- `required` - Value mustn't be empty
- `minlength` - Value has to match given minlength

Built-in Validation

Angular comes with the following built-in validation directives:

- `required` - Value mustn't be empty
- `minlength` - Value has to match given minlength
- `maxlength` - Value has to match given maxlength

Built-in Validation

Angular comes with the following built-in validation directives:

- `required` - Value mustn't be empty
- `minlength` - Value has to match given minlength
- `maxlength` - Value has to match given maxlength
- `pattern` - Value must match given RegExp pattern

Applying Validators

Validator directives can simply be applied to the form control element.

```
<input required minlength="3">
```

Validation State

`ngForm`, `ngModelGroup` and `ngModel` track validation state of form controls.

Validation State

`ngForm`, `ngModelGroup` and `ngModel` track validation state of form controls.

- **valid** - Form/Group/Control is valid

Validation State

`ngForm`, `ngModelGroup` and `ngModel` track validation state of form controls.

- **valid** - Form/Group/Control is valid
- **invalid** - Form/Group/Control is invalid

Validation State

`ngForm`, `ngModelGroup` and `ngModel` track validation state of form controls.

- **valid** - Form/Group/Control is valid
- **invalid** - Form/Group/Control is invalid
- **dirty** - Form/Group/Control has changed

Validation State

`ngForm`, `ngModelGroup` and `ngModel` track validation state of form controls.

- **valid** - Form/Group/Control is valid
- **invalid** - Form/Group/Control is invalid
- **dirty** - Form/Group/Control has changed
- **pristine** - Form/Group/Control hasn't changed

Validation State

`ngForm`, `ngModelGroup` and `ngModel` track validation state of form controls.

- **valid** - Form/Group/Control is valid
- **invalid** - Form/Group/Control is invalid
- **dirty** - Form/Group/Control has changed
- **pristine** - Form/Group/Control hasn't changed
- ...

```
<form #form="ngForm">

  <input ngModel name="firstname" required>

  <button [disabled]="!form.valid">Save</button>
</form>
```

```
<form #form="ngForm">

  <input ngModel name="firstname" required>

  <button [disabled]="!form.valid">Save</button>
</form>
```

Accessing Errors

Form control instances expose the error state on the `errors` property.

```
<input #firstname="ngModel" minlength="3">

<p *ngIf="firstname.errors.minlength">
  Whoops!
</p>
```

Accessing Errors

Form control instances expose the error state on the `errors` property.

```
<input #firstname="ngModel" minlength="3">

<p *ngIf="firstname.errors.minlength">
  Whoops!
</p>
```

Error Values

Different validators might expose different information about the error.

```
firstname.errors = {

  minlength: {
    actualLength: 2,
    requiredLength: 3
  },

  required: true
}
```

Exercise: Validation and Messages

Custom Validators

Custom Validators

A validator is a function that takes a `Control` and returns `null` or an error object.

```
function integerValidator(c: FormControl) {

  const REG_EXP = /^\-?\d+$/;
  var valid = REG_EXP.test(c.value);

  return (!valid) ? {
    integer: false
  } : null;
}
```

Validator Directive

We register validators via directives by supplying the `NG_VALIDATORS` provider.

```
@Directive({
  selector: '[validateInteger][ngModel]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useValue: integerValidator,
      multi: true
    }
  ]
})
class IntegerValidator {}
```

Validator Directive

We register validators via directives by supplying the `NG_VALIDATORS` provider.

```
@Directive({
  selector: '[validateInteger][ngModel]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useValue: integerValidator,
      multi: true
    }
  ]
})
class IntegerValidator {}
```

Multi Provider

Multi providers allow us to provide multiple dependencies for a single token.

```
@NgModule({
  ...
  providers: [
    { provide: 'MULTI_DEP', useValue: 'foo', multi: true },
    { provide: 'MULTI_DEP', useValue: 'bar', multi: true },
  ]
})
```

Multi Provider

Multi providers allow us to provide multiple dependencies for a single token.

```
@NgModule({
  ...
  providers: [
    { provide: 'MULTI_DEP', useValue: 'foo', multi: true },
    { provide: 'MULTI_DEP', useValue: 'bar', multi: true },
  ]
})
```

Multi Provider Injection

We can inject all registered values using the multi provider token.

```
@Component({
  ...
})
export class MyComponent {

  constructor(@Inject('MULTI_DEP') deps: []) {
    // deps == ['foo', 'bar']
  }
}
```

Built-in Multi Providers

Angular comes with several multi providers that we can extend with our custom code.

Built-in Multi Providers

Angular comes with several multi providers that we can extend with our custom code.

- **NG_VALIDATORS** Providers for validators

Built-in Multi Providers

Angular comes with several multi providers that we can extend with our custom code.

- **NG_VALIDATORS** Providers for validators
- **NG_ASYNC_VALIDATORS** Providers for async validators

Exercise: Custom Validator

Async Validators

Async Validator

Async validators return either a `Promise`, or an `Observable`.

```
checkEmailAvailability(contactsService: ContactsService) {
  return (c: FormControl) => {
    return contactsService.isEmailAvailable(c.value)
      .pipe(map(response => !response.error ? null : {
        emailTaken: true
      }));
  };
}
```

Async Validator

Async validators return either a `Promise`, or an `Observable`.

```
checkEmailAvailability(contactsService: ContactsService) {
  return (c: FormControl) => {
    return contactsService.isEmailAvailable(c.value)
      .pipe(map(response => !response.error ? null : {
        emailTaken: true
      }));
  };
}
```

Async Validator Directives

Validator Objects

If a validator has dependencies, we can create objects and take advantage of DI.

```
@Directive({
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: forwardRef(() => AsyncValidator),
      multi: true
    }
  ]
})
class AsyncValidator {

  constructor(contactsService: ContactsService) {}

  validate(c: FormControl) {}
}
```

```
@Directive({
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: forwardRef(() => AsyncValidator),
      multi: true
    }
  ]
})
class AsyncValidator {
  _validate: Function;

  constructor(contactsService: ContactsService) {
    this._validate = checkEmailAvailability(contactsService);
  }

  validate(c: FormControl) {}
}
```

```
@Directive({
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: forwardRef(() => AsyncValidator),
      multi: true
    }
  ]
})
class AsyncValidator {
  _validate: Function;

  constructor(contactsService: ContactsService) { ... }

  validate(c: FormControl) {
    return this._validate(c);
  }
}
```

Demo →

Exercise: Custom Async Validator

Reactive Forms

Reactive Forms

Reactive forms can be implemented using the following APIs:

Reactive Forms

Reactive forms can be implemented using the following APIs:

- **FormControl** - Represents a single form control

Reactive Forms

Reactive forms can be implemented using the following APIs:

- **FormControl** - Represents a single form control
- **FormGroup** - Represents a group of form controls

Reactive Forms

Reactive forms can be implemented using the following APIs:

- **FormControl** - Represents a single form control
- **FormGroup** - Represents a group of form controls
- **FormArray** - Similar to `FormGroup` but dynamic

Reactive Forms

Reactive forms can be implemented using the following APIs:

- **FormControl** - Represents a single form control
- **FormGroup** - Represents a group of form controls
- **FormArray** - Similar to `FormGroup` but dynamic
- **FormBuilder** - Factory API creating controls and control groups

Reactive Forms

Reactive forms can be implemented using the following APIs:

- **FormControl** - Represents a single form control
- **FormGroup** - Represents a group of form controls
- **FormArray** - Similar to `FormGroup` but dynamic
- **FormBuilder** - Factory API creating controls and control groups

Reactive forms are **easier to test**.

Activating reactive APIs

To activate reactive form APIs, we need to import **ReactiveFormsModule**

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

@NgModule({
  imports: [
    ...
    ReactiveFormsModule
  ]
  ...
})
export class ContactsModule {}
```

Activating reactive APIs

To activate reactive form APIs, we need to import **ReactiveFormsModule**

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

@NgModule({
  imports: [
    ...
    ReactiveFormsModule
  ]
  ...
})
export class ContactsModule {}
```

FormControl

Defines a part of a form and has its value and validity state (determined by optional validator).

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

@Component(...)
export class ContactCreatorComponent {
  firstname = new FormControl();
}
```

FormControl

Defines a part of a form and has its value and validity state (determined by optional validator).

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

@Component(...)
export class ContactCreatorComponent {
  firstname = new FormControl();
}
```

formControl Directive

We bind existing **FormControl**'s to a DOM element using **formControl**.

```
<input [formControl]="firstname">
```

Useful when using standalone input control **without** control group.

FormGroup

Defines a part of a form of fixed length, that contains other controls.

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

@Component(...)
export class ContactCreatorComponent {
  form = new FormGroup({
    firstname: new FormControl(),
    lastname: new FormControl()
  })
}
```

FormGroup

Defines a part of a form of fixed length, that contains other controls.

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

@Component(...)
export class ContactCreatorComponent {
  form = new FormGroup({
    firstname: new FormControl(),
    lastname: new FormControl()
  })
}
```

formGroup Directive

We bind existing **FormGroup** to a DOM element using **formGroup** directive.

```
<form [formGroup]="form">
  <input formControlName="firstname">
  <input formControlName="lastname">
</form>
```

FormGroup fields are bound using **formControlName**.

formGroup Directive

We bind existing **FormGroup** to a DOM element using **formGroup** directive.

```
<form [formGroup]="form">
  <input formControlName="firstname">
  <input formControlName="lastname">
</form>
```

FormGroup fields are bound using **formControlName**.

FormBuilder

FormBuilder provides convenient abstraction methods for all form control APIs.

FormBuilder

FormBuilder provides convenient abstraction methods for all form control APIs.

- **.group()** - Constructs a new control group

FormBuilder

FormBuilder provides convenient abstraction methods for all form control APIs.

- **.group()** - Constructs a new control group
- **.control()** - Constructs a new control

FormBuilder

FormBuilder provides convenient abstraction methods for all form control APIs.

- **.group()** - Constructs a new control group
- **.control()** - Constructs a new control
- **.array()** - Constructs a new control group from array

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

@Component(...)
export class ContactsCreatorComponent implements OnInit {
  form : FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.form = this.fb.group({
      firstname: '',
      lastname: ''
    });
  }
}
```

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

@Component(...)
export class ContactsCreatorComponent implements OnInit {
  form : FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.form = this.fb.group({
      firstname: '',
      lastname: ''
    });
  }
}
```

Applying Validators

Validator functions need to be added to a form model's field when building **reactive** forms.

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

...

ngOnInit() {
  this.form = this.fb.goup({
    firstname: ['', Validators.required],
    lastname: ['', Validators.required]
  });
}
```

Applying Validators

Validator functions need to be added to a form model's field when building **reactive** forms.

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

...

ngOnInit() {
  this.form = this.fb.goup({
    firstname: ['', Validators.required],
    lastname: ['', Validators.required]
  });
}
```

Composing Validators

Multiple validators need to be composed when building **reactive** forms.

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

...
ngOnInit() {
  this.form = this.fb.group({
    firstname: ['', [
      Validators.required,
      Validators.minlength(3)
    ]],
    lastname: ['', Validators.required]
  });
}
```

Composing Validators

Multiple validators need to be composed when building **reactive** forms.

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

...
ngOnInit() {
  this.form = this.fb.goup({
    firstname: ['', [
      Validators.required,
      Validators.minlength(3)
    ]],
    lastname: ['', Validators.required]
  });
}
```

Applying async validators

We add async validation as third argument when creating controls.

```
ngOnInit() {
  this.form = this.fb.goup({
    firstname: ['', syncValidator, asyncValidator],
    lastname: ''
  });
}
```

Exercise: FormBuilder

FormArray

FormArray

FormArray enables us to create **FormGroup**`s of unknown size.

FormArray

FormArray enables us to create **FormGroup**`s of unknown size.

- **FormArray.push(control: FormControl)** - adds new form control to collection
- **FormArray.removeAt(index: number)** - removes form control from collection by index

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

export class ContactsCreatorComponent implements OnInit {
  ...
  ngOnInit() {
    this.form = this.fb.group({
      ...
      phone: this.formBuilder.array([''])
    });
  }
}
```

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

export class ContactsCreatorComponent implements OnInit {
  ...
  ngOnInit() {
    this.form = this.fb.group({
      ...
      phone: this.formBuilder.array([''])
    });
  }
}
```

Adding fields

Form controls can be dynamically added using **push(control: FormControl)**

```
export class ContactsCreatorComponent implements OnInit {
  ...
  addPhoneField() {
    const control = <FormArray>this.form.get('phone');
    control.push(new FormControl(''));
  }
}
```

Adding fields

Form controls can be dynamically added using
push(control: FormControl)

```
export class ContactsCreatorComponent implements OnInit {
  ...
  addPhoneField() {
    const control = <FormArray>this.form.get('phone');
    control.push(new FormControl(''));
  }
}
```

Removing fields

Form controls can be removed using **removeAt(index: number)**

```
export class ContactsCreatorComponent implements OnInit {
  ...
  removePhoneField(index) {
    const control = <FormArray>this.form.get('phone');
    control.removeAt(index);
  }
}
```

Removing fields

Form controls can be removed using **removeAt(index: number)**

```
export class ContactsCreatorComponent implements OnInit {
  ...
  removePhoneField(index) {
    const control = <FormArray>this.form.get('phone');
    control.removeAt(index);
  }
}
```

```
<div formArrayName="phone">

  <div *ngFor="let phone of form.get('phone').controls;
               let i = index;">

    <md-input-container>
      <input placeholder="Phone" [formControlName]="i">
    </md-input-container>

    <button (click)="removePhoneField(i)">...</button>
    <button (click)="addPhoneField()">...</button>
  </div>

</div>
```

```
<div formArrayName="phone">

  <div *ngFor="let phone of form.get('phone').controls;
               let i = index;">

    <md-input-container>
      <input placeholder="Phone" [formControlName]="i">
    </md-input-container>

    <button (click)="removePhoneField(i)">...</button>
    <button (click)="addPhoneField()">...</button>
  </div>

</div>
```

ercise: FormArray for dynamic forms

Custom Form Controls

Custom Form Controls

Custom form controls provide functionality beyond native form controls. **Things to consider**:

Custom Form Controls

Custom form controls provide functionality beyond native form controls. **Things to consider**:

- Is there a native element with the same semantics?

Custom Form Controls

Custom form controls provide functionality beyond native form controls. **Things to consider**:

- Is there a native element with the same semantics?
- Can we rely on it and use CSS/progressive enhancement instead?

Custom Form Controls

Custom form controls provide functionality beyond native form controls. **Things to consider**:

- Is there a native element with the same semantics?
- Can we rely on it and use CSS/progressive enhancement instead?
- How can we make it accessible?

Custom Form Controls

Custom form controls provide functionality beyond native form controls. **Things to consider**:

- Is there a native element with the same semantics?
- Can we rely on it and use CSS/progressive enhancement instead?
- How can we make it accessible?
- How does it validate?

Creating custom controls

When creating custom form controls there are a couple of things we need to make sure:

Creating custom controls

When creating custom form controls there are a couple of things we need to make sure:

- It properly propagates changes to DOM/View

Creating custom controls

When creating custom form controls there are a couple of things we need to make sure:

- It properly propagates changes to DOM/View
- It properly propagates changes to Model

Creating custom controls

When creating custom form controls there are a couple of things we need to make sure:

- It properly propagates changes to DOM/View
- It properly propagates changes to Model
- Adds validity state

Creating custom controls

When creating custom form controls there are a couple of things we need to make sure:

- It properly propagates changes to DOM/View
- It properly propagates changes to Model
- Adds validity state
- It's accessible

ControlValueAccessor

Angular provides an interface **ControlValueAccessor** to implement custom form controls

- **writeValue(obj: any)** - Updates underlying DOM/View with new value
- **registerOnChange(fn: any)** - Registers handler to propagate changes to model
- **registerOnTouched(fn: any)** - Registers handler to propagate touched state

Custom address input

```
<fieldset formGroupName="address" fxLayout="column">
  <legend>Address</legend>
  <mat-form-field fxFlex>
    <input matInput placeholder="Street" formControlName="st
  </mat-form-field>
  ...
</fieldset>
```

should be:

```
<trm-address-input formControlName="address">
</trm-address-input>
```

```
@Component({
  selector: 'trm-address-input',
  providers: [






  ]
})
export class AddressInput {
  ...
}
```

```
@Component({
  selector: 'trm-address-input',
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => AddressInputComponent),
      multi: true
    }
  ]
})
export class AddressInput implements ControlValueAccessor {
  ...
}
```

```
export class AddressInput implements ControlValueAccessor {

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.form = this.fb.group({
      ...
    });
  }

  writeValue(value) {...}

  registerOnChange(fn) {...}

  registerOnTouched(fn) {...}
}
```

```
export class AddressInput implements ControlValueAccessor {

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.form = this.fb.group({
      ...
    });
  }

  writeValue(value) {...}

  registerOnChange(fn) {...}

  registerOnTouched(fn) {...}
}
```

writeValue()

writeValue() writes a new value from the form model into the view

```
export class AddressInput implements ControlValueAccessor {
  ...
  writeValue(value) {
    this.form.setValue(value)
  }
}
```

registerOnChange(fn)

registerOnChange() registers a function that can be later called to propagate changes from DOM/View

```
export class AddressInput implements ControlValueAccessor{
  propagateChange = (_: Address) => {};

  ngOnInit() {
    this.form.valueChanges.subscribe(value => {
      this.propagateChange(value);  // notify parent
    });
  }

  registerOnChange(fn) {
    this.propagateChange = fn;  // Cache callback to parent
  }
}
```

registerOnTouched(fn)

registerOnTouched() registers a function that can be later called to propagate touched state

```
export class AddressInput implements ControlValueAccessor {
  ...
  registerOnTouched(fn) {
    this.propagateTouched = fn;
  }
}
```

registerOnTouched(fn)

registerOnTouched() registers a function that can be later called to propagate touched state

```
<fieldset formGroupName="address" fxLayout="column">
  <legend>Address</legend>
  <mat-form-field fxFlex>

    <input matInput placeholder="Street"
           formControlName="street"
           (blur)="propagateTouched()" >

  </mat-form-field>
</fieldset>
```

registerOnTouched(fn)

registerOnTouched() registers a function that can be later called to propagate touched state

```
<fieldset formGroupName="address" fxLayout="column">
  <legend>Address</legend>
  <mat-form-field fxFlex>
    <input matInput placeholder="Street"
      (blur)="propagateTouched()" formControlName="street">
  </mat-form-field>
  ...
</fieldset>
```

Exercise: Custom Form Control

THOUGHTCAM

EXTEND YOUR MEMORY

- git add .
- git commit -am "(completed) - forms"
- git tag classroom/forms