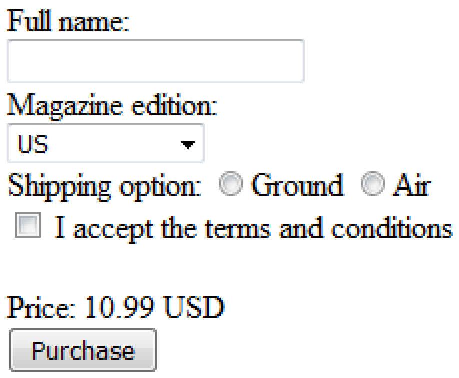
WEB AGE SOLUTIONS INC

Lab 1 - Model Driven Form

There are two ways to work with forms in Angular, template driven forms and model driven forms. Template driven forms use more code in the template to configure the form elements and link them to properties in the component class. A Model driven form uses JavaScript code in the component class to define and configure the form element properties and then the code in the template mainly just links form elements to the objects defined in the component.

In this lab we will explore form development using the model driven approach.

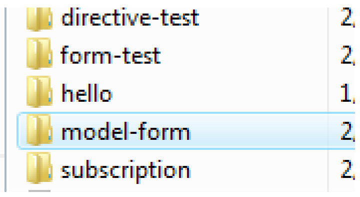
We will build a form that incorporates all the common HTML input controls. This will be the same form used in the template driven approach so you can see the similarities and differences between the two approaches. You will also add the same validation using the model driven approach.


Full name:
Magazine edition:
US
Shipping option: Ground Air
I accept the terms and conditions
Price: 10.99 USD
Purchase

Part 1 - Copy Existing Project

__1. In the '**C:\LabWork**' folder, copy the **form-test** folder.

__2. Paste and rename the folder to **model-form**.



Part 2 - Import ReactiveFormsModule Elements

The model driven approach is defined by classes in the 'ReactiveFormsModule' instead of the 'FormsModule' of the template driven style. There are also more classes imported into the component code that are then used to define model driven form objects.

In this section you will alter the existing project to use the required model driven definitions.

__1. In the **model-form/src/app** folder, open the existing '**app.module.ts**' file in a text editor.

__2. Modify the two places the '**FormsModule**' is referenced into '**ReactiveFormsModule**' as shown below.

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

@NgModule({
  imports:      [ BrowserModule, ReactiveFormsModule ],
  declarations:  [ Magazine ],
```

__3. Save and close the file after checking the changes above.

__4. In the **model-form/src/app/magazine** folder, open the existing '**magazine.component.ts**' file in a text editor.

__5. At the top of the component code, add the following line for new imports of the various model driven form definitions we will need in the component.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators,
          FormBuilder }  from '@angular/forms';

@Component({
    selector: "magazine",
```

Part 3 - Define Form Elements in Component

The key difference with model driven forms is that there are objects in the component that are created to represent the form elements using the Angular FormControl API.

__1. Scroll to the very bottom of the component code which currently defines several properties of the component.

__2. Modify the code as shown below in bold so that now these properties will contain various **FormControl** objects that have been initialized in the component code. Make sure not to miss the '**fullName**' property listed above the editions array.

```
  fullName = new FormControl('');
  editions = [
    {editionCode: 1, editionName: "US", price: "10.99 USD"},
    {editionCode: 2, editionName: "Canada", price: "14.99 CAD"},
    {editionCode: 3, editionName: "International", price: "23.99 USD"}
  ]
  selectedEdition = new FormControl(this.editions[0]); // Choose US ...
  selectedShipping = new FormControl('');
  acceptPolicy = new FormControl(false);
}
```

Note: The parameter to the **FormControl** constructor is the default value the form element will start with.

__3. After the code that you just modified, but within the curly brackets for the component definition, add the following property which will be a **FormGroup** object. This will be initialized next.

```
  selectedShipping = new FormControl('');
  acceptPolicy = new FormControl(false);

  magazineForm: FormGroup;

}
```

__4. After the code that you just modified, but within the curly brackets for the component definition, add the following method to initialize the form. Double check that you have the correct balance of parenthesis and curly brackets.

```
   magazineForm: FormGroup;

   createForm() {
      this.magazineForm = new FormGroup( {
          fullName: this.fullName,
          selectedEdition: this.selectedEdition,
          selectedShipping: this.selectedShipping,
          acceptPolicy: this.acceptPolicy
      } );
   }
}
```

__5. Go to the beginning of the Magazine component class definition and add the following code for a constructor that will call the '**createForm**' method you just added.

```
export class Magazine {
   constructor()  {
      this.createForm();
   }

   submitForm() {
```

__6. Modify the code of the '**submitForm**' method to extract the values from the magazine form since the '**magazineForm.value**' now represents the data that is filled in on the form.

```
  }

  submitForm() {
    let requestData = {
       customerName: this.magazineForm.value.fullName,
       productCode: this.magazineForm.value.selectedEdition.editionCode,
       acceptPolicy: this.magazineForm.value.acceptPolicy,
       shipMode: this.magazineForm.value.selectedShipping
    }
    alert(JSON.stringify(requestData))
  }

  fullName = new FormControl('');
```

__7. Save the file but leave it open for the next section.

Part 4 - Modify Component Template

Right now the template for the magazine component is still using template driven forms. There will be several changes needed to turn the HTML template into something appropriate for a model driven form. The good thing is the code will generally be much simpler in the template since more is done in the component class. This section will remove all of the current template and build it back up with the model driven approach.

__1. Make sure you still have the '**magazine.component.ts**' file open in a text editor.

__2. Find the @Component decorator and delete the entire content of the '**template**' property, leaving only a blank line or two between the back ticks. The bold code below shows what the property should look like when you are done clearing it making sure to still have an opening and closing back tick.

```
@Component({
    selector: "magazine",
          styles: ["input.ng-touched.ng-invalid {background: red}",
        "input:required {box-shadow:none;}"],
    template: `

    `
})
```

__3. Within the back ticks, add the following form tag. Make sure to also have the closing form tag with some extra space to add other elements in the form later. Note the '**[formGroup]**' attribute and the name of the FormGroup property from the component.

```
template: `
<form [formGroup]="magazineForm" (ngSubmit)="submitForm()">

</form>
`
```

Note: One difference here is that were are linking the 'submitForm' method to the submit event of the entire form. In the component driven form it was linked to clicking a button. This was mainly because for some template driven form elements we didn't need the surrounding form tag. Here we are using it right away to link to the FormGroup model in the component.

__4. Add the label and textBox for the '**fullName**' control. Notice the value of the '**[formControl]**' attribute is the name of the form element from the FormGroup model.

```
<form [formGroup]="magazineForm" (ngSubmit)="submitForm()">
    Full name:<br/>
    <input [formControl]="fullName" type="text" /><br/>

</form>
```

__5. Add the following code for the drop-down for the magazine edition.

```
    <input [formControl]="fullName" type="text" /><br/>
    Magazine edition:<br/>
    <select [formControl]="selectedEdition" >
        <option *ngFor="let e of editions"
            [ngValue]="e">{{e.editionName}}</option>
    </select><br/>

</form>
```

Note: In this code the <**option**> tag is exactly like it was with the template driven form.

You still use the '*ngFor' directive to iterate over the options and '[ngValue]' for what each option value should be.

__6. Add the following code for the shipping option radio buttons.

```
            [ngValue]="e">{{e.editionName}}</option>
    </select><br/>
    Shipping option:
    <input type="radio" [formControl]="selectedShipping"
         value="GROUND"/>Ground
    <input type="radio" [formControl]="selectedShipping"
         value="AIR"/>Air <br/>

</form>
```

__7. Add the code for the accept terms checkbox.

```
    <input type="radio" [formControl]="selectedShipping"
         value="AIR"/>Air <br/>
    <input [formControl]="acceptPolicy" type="checkbox" />
        I accept the terms and conditions<br/>
    <br/>

</form>
```

__8. Finally, add the code to display the price and the submit button at the bottom of the form.

```
        I accept the terms and conditions<br/>
    <br/>
    Price: {{magazineForm.value.selectedEdition.price}}
    <br/>

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

__9. Double check that your entire 'template' property appears as shown below.

```
template: `
<form [formGroup]="magazineForm" (ngSubmit)="submitForm()">
    Full name:<br/>
    <input [formControl]="fullName" type="text" /><br/>
    Magazine edition:<br/>
    <select [formControl]="selectedEdition" >
        <option *ngFor="let e of editions"
            [ngValue]="e">{{e.editionName}}</option>
    </select><br/>
    Shipping option:
    <input type="radio" [formControl]="selectedShipping"
         value="GROUND"/>Ground
    <input type="radio" [formControl]="selectedShipping"
         value="AIR"/>Air <br/>
    <input [formControl]="acceptPolicy" type="checkbox" />
        I accept the terms and conditions<br/>
    <br/>
    Price: {{magazineForm.value.selectedEdition.price}}
    <br/>

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

__10. Save the file but leave it open in the text editor.

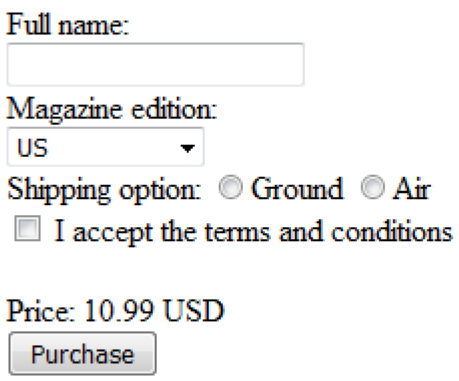
Part 5 - Test Basic Form

Now that you have the basic elements of the form implemented in the model driven style, you can test the form. This will verify it works before adding more complex things like Validation.

__1. Open a command prompt and go to the root folder of the project **C:\LabWork\model-form**.

__2. Run the following command.

```
npm run start
```


Full name:
Magazine edition:
US
Shipping option: Ground Air
I accept the terms and conditions
Price: 10.99 USD
Purchase

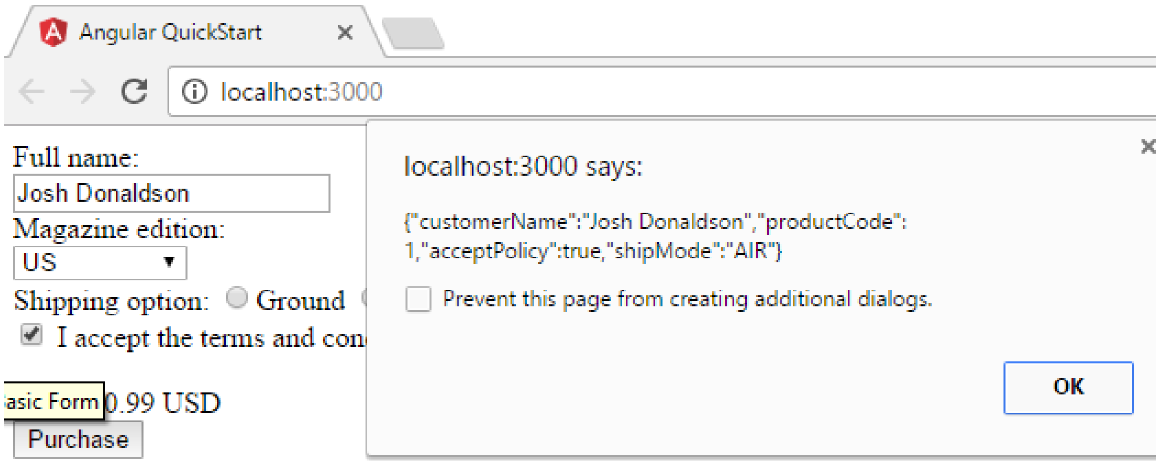
__3. Enter a name in the full name field.

__4. Change the magazine edition. Verify that the price display changes correctly.

__5. Select one of the shipping options.

__6. Check the box to accept terms.

__7. Click the **Purchase** button.



localhost:3000 says:

```
{"customerName":"Josh Donaldson","productCode":1,"acceptPolicy":true,"shipMode":"AIR"}
```

__8. Verify that the JSON has the correct user input. If the browser has an option to prevent further dialogs make sure to NOT select the option. Click OK.

__9. Switch the drop-down to a different magazine edition and check that the 'productCode' property in the resulting JSON is still set correctly.

__10. Leave the browser and page open for the future.

Part 6 - Simplify Form Initialization

Currently you initialize the form by creating each FormControl object separately and the manually adding them to the FormGroup. Using the FormBuilder component there is an easier way to initialize the form which you will do in this section.

__1. In the '**magazine.component.ts**' file, find the code that creates the individual FormControl objects and completely remove the entire property and FormControl constructor from the code as shown in strike-through below.

```
~~fullName = new FormControl('');~~
editions = [
  {editionCode: 1, editionName: "US", price: "10.99 USD"},
  {editionCode: 2, editionName: "Canada", price: "14.99 CAD"},
  {editionCode: 3, editionName: "International", price: "23.99 USD"}
]
~~selectedEdition = new FormControl(this.editions[0]); //Choose US ...~~
~~selectedShipping = new FormControl('');~~
~~acceptPolicy = new FormControl(false);~~
```

__2. Find the code for the '**createForm**' method and make the following changes shown below in bold. This will call a 'group' method of the 'formBuilder' object and change what is used for the values of the properties of the object passed as the parameter to the method. Note that the empty string ('') might be tough to see in bold but it is used in two places.

```
  createForm() {
    this.magazineForm = this.formBuilder.group({
      fullName: '',
      selectedEdition: this.editions[0],
      selectedShipping: '',
      acceptPolicy: false
    });
  }
```

Note: The 'FormBuilder.group()' method will take an object as a parameter and will take the value of every property of that object and construct a FormControl object from it. This change is simply eliminating the intermediary properties in the component that had initialized and stored the FormControl objects individually.

__3. Find the constructor of the Magazine component class and add a parameter as shown below. This will have Angular supply a FormBuilder object when constructing the component and store it as a field in the component. This is required in order to be able to call the 'group' method to initialize the form.

```
export class Magazine {
    constructor(private formBuilder: FormBuilder)  {
        this.createForm();
    }
```

__4. In the component template, find all occurrences of '[formControl]' and replace them with '**formControlName**'. The name of this attribute directive shouldn't have any square brackets or quotes. There should be 5 places to perform a replacement.

```
<input formControlName="fullName" type="text" /><br/>
Magazine edition:<br/>
<select formControlName="selectedEdition" >
    <option *ngFor="let e of editions"
    ...
Shipping option:
<input type="radio" formControlName="selectedShipping"
     value="GROUND"/>Ground
<input type="radio" formControlName="selectedShipping"
     value="AIR"/>Air <br/>
<input formControlName="acceptPolicy" type="checkbox" />
    I accept the terms and conditions<br/>
```

Note: This change is required because using the '[formControl]' syntax is only possible when the FormControl objects are declared as individual properties of the component. Now that the form is assembled more dynamically, the 'formControlName' directive needs to be used.

__5. Save the file but leave it open in the text editor.

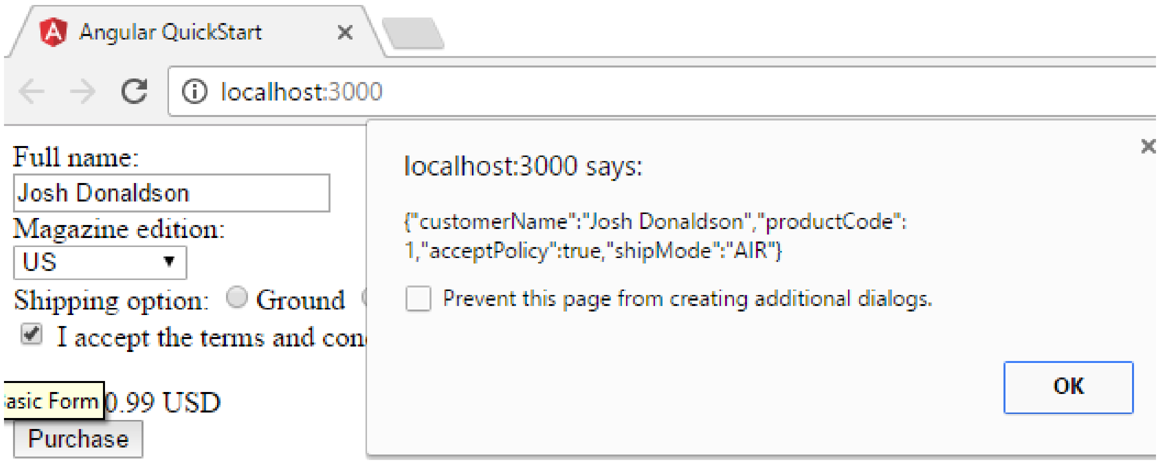
Part 7 - Test Simplified Form Initialization

Although you haven't changed the definition or behavior of the form, it will be good to test and make sure things are correct.

__1. Return to the command prompt you had open in the **model-form** folder.

__2. Check there are no issues recompiling the code and the web browser should refresh automatically.

__3. Verify the form still functions the same as before, even though you are initializing the form differently.


Angular QuickStart

localhost:3000

Full name: Josh Donaldson

Magazine edition: US

Shipping option: Ground

I accept the terms and con

asic Form 0.99 USD

Purchase

localhost:3000 says:

{"customerName":"Josh Donaldson","productCode":1,"acceptPolicy":true,"shipMode":"AIR"}

Prevent this page from creating additional dialogs.

OK

Part 8 - Add Form Validation

Just like with template driven forms, model driven forms can have validation logic. More of this logic is in the code of component although you can also use various form object properties in the HTML template of the component to do things like disable buttons of invalid forms or display error messages next to incorrect form fields.

__1. In the '**magazine.component.ts**' file, find the code that calls the 'FormBuilder.group' method.

__2. Modify the way the '**fullName**' property in the parameter object is initialized. This changes it from just a simple empty string to an array with two elements, the same empty string and a '**Validators.required**' value for the validation constraints for the FormControl object to be constructed.

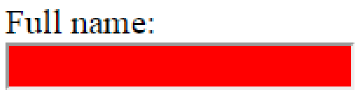
```
createForm() {
  this.magazineForm = this.formBuilder.group({
    fullName: ['', Validators.required ],
    selectedEdition: this.editions[0],
```

__3. Save the changes to the component.

__4. Check there are no issues recompiling the code and the web browser should refresh automatically.

__5. Make sure that the full name field is shown using the usual white background.

__6. Click inside the text box and immediately click outside. Now the background should change to red.



Note: The styles are being applied because you left the '**styles**' property of the component the same and only changed the 'template' code.

__7. Fill in some kind of value for the name and verify the text box shows up in white again.

__8. Test that the rest of the form still functions as expected.

Part 9 - Use Form Validation Status Properties

Just like with template driven forms, model driven forms have various status properties that can be used to dynamically modify the display or behavior of the form.

__1. In the '**magazine.component.ts**' file, find the code for the component template.

__2. Add the following text and expressions to display various validation properties of the form.

```
Full name:<br/>
<input formControlName="fullName" type="text" /><br/>
Valid: {{magazineForm.controls.fullName.valid}}<br/>
Touched: {{magazineForm.controls.fullName.touched}}<br/>
Dirty: {{magazineForm.controls.fullName.dirty}}<br/>
Magazine edition:<br/>
<select formControlName="selectedEdition" >
```

__3. Between the label for the name textbox and the textbox itself, add the following '**<span>**' tag that will be hidden based on some attributes of the displayed form.

```
    <form [formGroup]="magazineForm" (ngSubmit)="submitForm()">
        Full name:<br/>
        <span style="color: red"
            [hidden]="magazineForm.controls.fullName.valid ||
                magazineForm.controls.fullName.untouched">
        Please enter your name<br/></span>
        <input formControlName="fullName" type="text" /><br/>
        Valid: {{magazineForm.controls.fullName.valid}}<br/>
```

__4. Scroll down to the very bottom of the component template and add the following to the submit button to disable it if the form is not valid.

```
        <br/>

        <button [disabled]="magazineForm.invalid"
            type="submit">Purchase</button>
    </form>
```

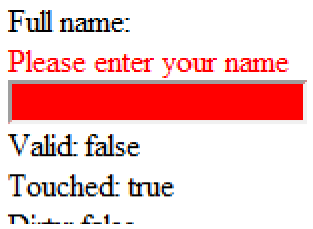
__5. Save the changes to the component.

__6. Check there are no issues recompiling the code and the web browser should refresh automatically.

__7. Make sure that the error message is not displayed by default.

__8. Check that the 'Purchase' submit button is disabled by default.

__9. Click the full name text box and immediately click outside. You should see the error message. Also check the form validation properties displayed by text are changing.


Full name:
Please enter your name
Valid: false
Touched: true

__10. Start typing into the full name text box. The **Purchase** button should become enabled and the error message should go away.

__11. Check that the rest of the function of the form still works as before.

__12. In the command prompt, hit '**<CTRL>-C**' to terminate the batch job.

__13. Close all open files.

Part 10 - Review

In this lab you saw model driven forms. By implementing the same form as in the template driven form lab, you saw some of the differences between the two styles. With a model driven form there is more in the code of the component and a (sometimes small) reduction in the code of the component's HTML template.