docs/edit/master/content/docs/v3/developer-resources/forms/index.md)

Ionic and Forms

Forms have gone through quite a bit of a change in Angular 2, but all for the best. Apps can have much more complicated and in depth forms than were allowed in Angular 1, and the new Forms module allows for this. Let’s look at three distinct examples of forms and when you would want to use each one.

Forms with `[(ngModel)]`

Simple forms that binds and passes an object in our class

Pros: Simpler API, more familiar to Angular 1 developers.

Cons: Less programmatic control, hard to unit test

```
import { Component } from '@angular/core';
@Component({
  template: `
    <form (ngSubmit)="logForm()">
      <ion-item>
        <ion-label>Todo</ion-label>
        <ion-input type="text" [(ngModel)]="todo.title" name="title"></ion-input>
      </ion-item>
      <ion-item>
        <ion-label>Description</ion-label>
        <ion-textarea [(ngModel)]="todo.description" name="description"></ion-textare
      </ion-item>
      <button ion-button type="submit" block>Add Todo</button>
    </form>
  `,
})
export class FormsPage {
  todo = {}
  logForm() {
    console.log(this.todo)
  }
}
```

This is very similar to how forms were made in Angular 1/Ionic 1 and will be the most familiar to other developers.

Note: If you use ngModel within a Form tag, you have to provide a `name` property. If you do not, you must set `standalone` to true in ngModelOptions.

```
<!-- with name set -->
<ion-item>
  <ion-label>Todo</ion-label>
  <ion-input type="text" [(ngModel)]="todo.title" name="title"></ion-input>
</ion-item>

<!-- or with standalone set to true -->
<ion-item>
  <ion-label>Todo</ion-label>
  <ion-input type="text" [(ngModel)]="todo.title" [ngModelOptions]="{standalone: true
</ion-item>
```

Forms with FormBuilder

Forms that are created in our class. The logic for the form is held entirely in the class

Pros: Easier to unit test, one place to go to for modifications

Cons: Must import FormBuilder

```
import { Component } from '@angular/core';
import {Validators, FormBuilder, FormGroup } from '@angular/forms';
@Component({
  template: `
    <form [formGroup]="todo" (ngSubmit)="logForm()">
      <ion-item>
        <ion-label>Todo</ion-label>
        <ion-input type="text" formControlName="title"></ion-input>
      </ion-item>
      <ion-item>
        <ion-label>Description</ion-label>
        <ion-textarea formControlName="description"></ion-textarea>
      </ion-item>
      <button ion-button type="submit" [disabled]="!todo.valid">Submit</button>
    </form>
  `
})
export class FormsPage {
  private todo : FormGroup;

  constructor( private formBuilder: FormBuilder ) {
    this.todo = this.formBuilder.group({
      title: ['', Validators.required],
      description: [''],
    });
  }
  logForm(){
    console.log(this.todo.value)
  }
}
```

This approach puts most of your form logic in the class for your component. This makes testing much easier, and the form itself becomes much easier to read.

Forms with Templates

Forms are not driven by an object in our class, but by an actual reference to the form itself

Pros: More vanilla HTML forms, all of the logic is in the template

Cons: More difficult to test, template can be cluttered.

```
import { Component } from '@angular/core';
@Component({
  template: `
    <form #form="ngForm" (ngSubmit)="logForm(form)" novalidate>
      <ion-item>
        <ion-label>Todo</ion-label>
        <ion-input type="text" required [(ngModel)]="todo.title" ngControl="title"></
      </ion-item>
      <ion-item>
        <ion-label>Description</ion-label>
        <ion-textarea [(ngModel)]="todo.description" ngControl="description"></ion-te
      </ion-item>
      <button ion-button type="submit" block>Add Todo</button>
    </form>
  `,
})
export class FormsPage {
  todo = {
    title: '',
    description: ''
  };
  logForm(form) {
    console.log(form.value)
  }
}
```

This is very similar to how you can use forms with just ngModel, but instead of using the actual model of the form, we can use the reference to the form itself. In the first example, all of our values were coming directly from the model `todo`. With template driven forms, the values are pulled directly from the form itself.

Parting words

While all methods of creating forms work with Ionic and its components, we strongly recommend using the FormBuilder API as it allows for a much similar method, and is easily testable with unit tests.