# 1Angular 2.0: Forms

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

## 

## Objective

Angular 1.x uses two-way data bindings to get data from the DOM. This can create cycles in the change detection graph, which results in the following problems:

* It is difficult to reason about the data flow in the application and understand which components can affect which values.
* The framework has to run digest multiple times, which is not performant.
* Because of multiple digests, it is impossible to determine and notify the component that the model is stable. This is because notification can further change data, which can restart the binding process.
* The framework cannot set all the bindings on a parent element before processing its children, which makes implementing certain patterns very difficult.
* The framework is harder to learn. The developer has to think about the digest cycle.
* Since the order of updates is not well defined, troubleshooting certain types of problems is challenging.

Instead of a generic mechanism for two-way data bindings, Angular 2 can have a more constrained API for getting data from the DOM as a remedy for these issues.

## Proposed Design

* FormControl, FormControlGroup, FormControlArray are three basic primitives for constructing forms.
* FormBuilder provides a DSL making form declaration more concise. FormBuilder creates FormControl, FormControlGroup, FormControlArray.
* Forms can be declared in the template of a component. This creates FormControl, FormControlGroup, FormControlArray.

Even though most developers will use FormBuilder or will declare their forms in the template, most examples in the document use FormControl, FormControlGroup, FormControlArray directly. This is to make the examples more explicit.

### Simple Forms

|  |  |
| --- | --- |
| <form [control-group]='loginForm'>  <input control='login'>  <input control='password' type="password">  </form> | class SomeComponent {  const FormControl("login"),  new FormControl("password")  ]);  }  } |

### Nested Forms

FormControlGroup implements the FormControl interface. This enables the nesting of control groups.

|  |  |
| --- | --- |
| <form [control-group]='personInfoForm'>  <div control-group='address'>  <input control='city'>  <input control='country'>  </div>  <div control-group='name'>  <input control='first'>  <input control='last'>  </div>  </form> | class SomeComponent {  constructor() {  this.personInfoForm = new FormControlGroup("form", [  new FormControlGroup("address", [  new FormControl("city"),  new FormControl("country")  ]),  new FormControlGroup("name", [  new FormControl("first"),  new FormControl("last")  ])  ]);  }  } |

**Arrays**

We can create multiple instances of the same control using FormControlArray.

|  |  |
| --- | --- |
| <form [control-group]='itemsForm'>  <input template="  ng-repeat: #item in itemsForm.items" control='item'>  </form> | class SomeComponent {  constructor() {  this.itemsForm = new FormControlGroup("form", [  new FormControlArray("items", new FormControl("item"))  ]);  }  } |

FormControl, FormControlGroup, FormControlArray are not singletons, and do not match particular DOM nodes, when they are constructed by the component. They are the description of what a form should look like.

**Bound and Unbound**

A FormControl can be in two states: bound and unbound. We have to have the two states because when we create a

form object in the constructor of a component, we have not compiled the template yet. So we have not bound the form to the DOM yet. So only bound controls have DOM elements and values. Also, only bound controls affect their form validity.

Have a look at the following example:

|  |  |
| --- | --- |
| <form [control-group]='form'>  <div template="ng-if: someExp">  <input control='item'>  </div>  </form> | class SomeComponent {  constructor() {  this.form = new FormControlGroup("form", [  new FormControl("item")  ]);  }  } |

* The form is unbound in the constructor. This is because we have not compiled the template of that component yet.
* During the compilation of the template, the input control and the form itself get bound to DOM elements.
* When the ng-if removes the input element from the DOM, the corresponding control gets unbound again. At that point the form is bound, but it has no bound children.

|  |
| --- |
| //when item is bound  this.form.controls.item.bound === true  this.form.controls.item.value === "some value"  this.form.value === {item: "some value"}  this.form.valid //depends on the item’s validity  //when item is unbound  this.form.controls.item.bound === false  this.form.controls.item.value === undefined  this.form.value === {}  this.form.valid //DOES NOT depend on the item’s validity |

### Alternative designs

An alternative design is to separate bound and unbound states into two separate classes (e.g., ProtoFormControl and FormControl.) Although it separates the concerns better, it makes the API rather awkward:

|  |
| --- |
| class SomeComponent {  constructor() {  this.form = new ProtoFormControlGroup("form", [  new ProtoFormControl("item")  ]);  }  this.form.onBound((boundForm) => {  boundForm.onChange(() => {});  });  } |

Another alternative is to introduce OptionalControl decorator.

|  |  |
| --- | --- |
| <form [control-group]='form'>  <div template="ng-if: form.controls.item.included">  <input control='item'>  </div>  </form> | class SomeComponent {  constructor() {  this.form = new FormControlGroup("form", [  new OptionalControl(new FormControl("item"))  ]);  }  } |

Where OptionalControl has the following interface:

|  |
| --- |
| interface OptionalControl extends FormControl {  included:boolean;  } |

Only included optional controls are used to run validations or generate the form’s value.

## Interfaces

Let's look at the operations of the FormControl interface.

|  |
| --- |
| interface FormControl {  get value();  set value(newValue);  bind(element);  get bound:boolean;  name:string;  valid:boolean;  errors:Map;  onChange(fn);  clone():FormControl;  }  interface FormControlGroup extends FormControl {  writeTo(object);  readFrom(object);  get controls:Map<String,FormControl>;  } |

### Getting Controls

|  |
| --- |
| var form = new FormControlGroup("form", [  new FormControl("login"),  new FormControl("password")  ]);  var login = form.controls["login"]; |

In Javascript we can use the dot notation:

|  |
| --- |
| var login = form.controls.login; |

### Reading and Writing Values

|  |
| --- |
| var form = new FormControlGroup("form", [  new FormControl("login"),  new FormControl("password")  ]);  var loginValue = form.controls.login.value;  form.controls.login.value = 'New Login';  var formValue = form.value; //will construct an object {login: "some value", password: "some value"};  form.value = {login:"new value", password: 'new value'}; //will update the login and password controls |

If constructing value objects is too expensive, or there is an existing domain object, we can use readFrom and writeTo instead.

|  |
| --- |
| form.readFrom(person);  form.writeTo(person); |

We can also use `value =` or `readFrom` to initialize a form.

|  |
| --- |
| this.form = new FormControlGroup("form", [  new FormControl("login"),  new FormControl("password")  ]);  this.form.value = {login : "some silly default"}; |

### Listening to Changes

There are many ways to interact with a form. For instance, we can set up an event handler that will use its value.

|  |  |
| --- | --- |
| <form [control-group]='loginForm' (submit)="submitForm()">  <input control='login'>  <input control='password' type="password">  </form> | class SomeComponent {  submitForm() {  submitFormService.submit(this.loginForm.value);  }  } |

There are cases, however, when we want to know if the values of any of the controls in a form changed. We can do that by attaching an event listener to the form object itself.

|  |
| --- |
| form.onChange(() => {  //do something  }); |

Every FormControl is observable, so we can listen to the changes in a particular control or a group of controls. When a FormControl gets updates, it notifies its listener. This notification happens synchronously and does not involve Angular’s change detection. A FormControlGroup notifies its listeners when one of its controls changes.

### Validations

A FormControl can have a validator assigned to it.

|  |
| --- |
| this.form = new FormControlGroup("form", [  new FormControl("login", Validations.required),  new FormControl("password")  ]); |

The validator runs when the control's value changes. By default, the valid and errors properties of a FormControlGroup is calculated by reducing the valid and errors properties of its children.

|  |
| --- |
| this.form.controls.login.errors; returns {required: ["must be present"]}  this.form.errors; //returns {login: {required: ["must be present"]}, password: {}} |

This behaviour can be overridden.

|  |
| --- |
| this.form = new FormControlGroup("form", [  new FormControl("password"),  new FormControl("password-confirmation")  ], samePassword);  var isFormValid = this.form.valid;  this.form.errors: {base: ["password are not the same"]} |

From the control’s perspective, the validator is just a pure function. For instance:

|  |
| --- |
| function required(c:FormControl):Map<String, List<String>>{  if(! c.value)  return {required: ["must be present"]);  } |

A variety of combinators will be provided to compose validators.

|  |
| --- |
| function compose(validators:List) {  return function(c:FormControl):Map<String,List<String>>{  return validators.reduce((m, v) => mergeResult(m, v), {});  }  }  var validator = compose(required, maxLength(10)) |

This API is not completely fleshed out.

### Using FormBuilder (DSL for Defining Forms)

Using FormControlGroup and FormControl directly can be verbose. To make form configuration more compact there will be a DSL built on top the basic primitives.

|  |  |
| --- | --- |
| this.form = new FormControlGroup("form", [  new FormControl("login", Validations.required),  new FormControl("password")  ]); | this.form = formBuilder.group({  login: {type: string, validator: Validations.required},  password: {type: string}  }); |

We can also use the builder to instantiate individual controls:

|  |  |
| --- | --- |
| var loginControl = new FormControl("login", Validations.required); | var loginControl = formBuilder.control("login", Validations.required) |

In addition to making the configuration more concise, FormBuilder allows replacing the form with a test double.

The details of the DSL have not been fleshed out yet.

**Using DI**

DI can be very useful when constructing validators.

Let's say we have the following validator:

|  |
| --- |
| class UniqEmailValidator {  constructor(http:Http) {}  validate(object){}  } |

Since it is stateless, we can inject it into components.

|  |
| --- |
| class SomeComponent {  constructor(uniqEmailValidator:UniqEmailValidator){  this.signupForm = new FormControlGroup("form", [  new FormControl("login", uniqEmailValidator),  new FormControl("password")  ]);  }  } |

Such validators can be injected together as a group.

|  |
| --- |
| class CustomValidators {  uniqEmail:Validator;  constructor(uniqEmail:UniqEmailValidator) {  this.uniqEmail = uniqEmail;  }  }  class SomeComponent {  constructor(validators:CustomValidators){  this.signupForm = new FormControlGroup("form", [  new FormControl("login", validators.uniqEmail),  new FormControl("password")  ]);  }  } |

Finally, when it does not suffice, a factory pattern can be applied.

|  |
| --- |
| class FormFactory {  constructor(http:Http) {  this.http = http;  }  uniqEmailControl(name:string) {  return new FormControl(name, new UniqEmailValidator(http));  }  control(name:string) {  return new FormControl(name);  }  }  class SomeComponent {  constructor(ff:FormFactory) {  this.signupForm = new FormControlGroup("form", [  ff.uniqEmailControl("login"),  ff.control("password")  ]);  }  } |

### Using Custom Elements

There are three things that a FormControl needs to interact with a DOM element: a list of events the element can emit, a function that reads a value of the element, a function that updates the value of the element. We can override these functions to work with custom elements.

|  |
| --- |
| var customControl = new FormControl("custom-login", validator, {  events: ['custom1', 'custom2'],  readValue: (el) => el.customValue,  writeValue: (el, value) => el.customValue = value,  }) |

When this is not enough, we can provide a custom implementation of the FormControl interface.

### Debounce

|  |
| --- |
| var control = new FormControl("slow-input", validator, {  debounce: {"default" : 500, "blur" : 0}  }) |

The slow-input control will update the model in 500 milliseconds after an event gets fired, unless it is a blur event.

When this is not enough, we can provide a custom implementation of the FormControl interface.

### Decorators

|  |  |
| --- | --- |
| <input control="price" price-decorator> | @Decorator({  selector: "input[price-decorator]"  })  class PriceDecorator {  constructor(fc:FormControl) {  fc.formatters.push(formatPrice);  fc.parsers.push(parsePrice);  }  } |

A decorator can add/remove parsers, formatters, and validators.

### Forms Declaration in Html

Often we do not want to burden the controller with the knowledge about a form structure. If the form is simple enough, we may want to express it all in html.

|  |  |
| --- | --- |
| <form #login new-control-group (submit)="submitForm(login.form)">  <input control='login'>  <input control='password' type="password">  </form> | class SomeComponent {  submitForm(form:FormControl) {  submitFormService.submit(form.value);  }  } |

## Here instead of using a FormControl that is created by the controller, the framework will create it when instantiating the view. This control then can be passed into functions such as submitForm.

You can provide the initial state for the form defined in html as follows:

|  |  |
| --- | --- |
| <form var=”self” [new-control-group]="{login: 'some init value'}" (submit)="submitForm(self.form)">  <input control='login'>  <input control='password' type="password">  </form> | |

## 

### Forms, Events, and Dirty-Checking

When a FormControl gets bound, it will attach an event listener to its DOM element. So when the DOM element fires an event, the FormControl will update its value. This includes running its validator and notifying its listeners. The form itself does not trigger dirty-checking cycles. To do that we need to create a binding. One way to do that is to write the form’s value into another object.

|  |  |
| --- | --- |
| class SomeComponent {  someEventHandler() {  this.domainObject = this.form.value;  }  } | {{domainObject.value}} |

or

|  |  |
| --- | --- |
| class SomeComponent {  someEventHandler() {  this.form.writeTo(this.domainObject);  }  } | {{domainObject.value}} |

Of course we can always bind to the form object itself, although it is discouraged:

|  |
| --- |
| {{form.controls.login.value}} |

When the value of a FormControl is set by the user, the control will schedule a DOM write to update the DOM.

**More Examples**

## Account Registration Form

This is is a complex form that demonstrates the following features:

* Local validations (required)
* Cross-control validations (same)

|  |  |
| --- | --- |
| <form [control-group]='form'>  <div control-group='loginInfo'>  <input control='login'>  <input control='password'>  <input control='pconfirm'>  </div>  <div>  <div template="ng-repeat: #c in: form.contacts">  <div template="ng-if: c.isPhone">  Phone: <input [control]="c">  </div>  <div template="ng-if: c.isEmail">  Email: <input [control]="c">  </div>  </div>  <button on-click=addPhone()'>  <button on-click=addEmail()'>  </div>  </form> | class SomeComponent {  constructor() {  this.form = new FormControlGroup("form", [  new FormControlGroup("loginInfo", [  new FormControl("login", Validations.required),  new FormControl("password", Validations.required),  new FormControl("pconfirm", Validations.required)  ], Validations.same("password", "pconfirm")),  new FormControlArray("contacts", new FormControl())  ]);  }  addPhone(){  this.form.controls.contacts.push(new Contact("phone"));  }  addEmail(){  this.form.controls.contacts.push(new Contact("email"));  }  } |

* Using nested groups (loginInfo)
* Using arrays of controls (contacts)

|  |  |
| --- | --- |
|  |  |

## Implementation

There are prototypes implementing most of the functionality described in the document. Once we agree on the API, I will add implementation notes here.