Forms and ngModel Design in 1.x

*Status: (Draft)*

*Authors:* [*pete@baconarwin.com*](mailto:pete@baconarwin.com)*,* [*mjstaffa@gmail.com*](mailto:mjstaffa@gmail.com)

*This document is published to the web as part of the public* [*Angular 1.x Design Docs*](https://drive.google.com/open?id=0B7Ovm8bUYiUDZkNjZ0NscWlaODg&authuser=0) *folder*

# Objective

Refactor the ngModel conceptual model to create a more clear intuitive API for both working with model data that connects to input controls in forms and for writing input directives.

# Background

The complexity of the forms implementation in Angular 1.x has grown considerably over the last 12 months. There are now many moving parts; and consequently many corner cases.

In particular the asynchronicity has made it increasingly difficult to understand and maintain how the different parts fit together. Moreover there are some semantic discussions about what role validity has to play in how and when the model and view values are updated.

# Prior Art

How do other js frameworks / libraries do input manipulation / validation? Is there anything good anyway?

## Angular Libraries

# [angular-formly](https://github.com/formly-js/angular-formly)

# [angular-schema-form](https://github.com/Textalk/angular-schema-form)

# [mongoose js](http://mongoosejs.com/) (for schemas, which translate well enough to form specs) and validation errors. forms-angular makes use of both.

* [ng-schema](https://github.com/dalcib/ng-schema)

## Angular 2

We also should consider the new (not prior) Angular 2.0 Forms design doc: <https://docs.google.com/document/d/1US9h0ORqBltl71TlEU6s76ix8SUnOLE2jabHVg9xxEA/edit#heading=h.s3llc14qem13>

## Other Libraries

* <http://parsleyjs.org/> validation library
  + Uses a priority for each validation rule that orders which validators to run
  + It stops validation as soon as one rule fails
  + Supports specifying the event that triggers the validation
  + Supports validation between fields by referencing via HTML ids
* Dijit/Form/form from The Dojotoolkit (dojotoolkit.org)

# High Level Design

## Current Design Problems

The behaviour of ngModel, forms, inputs validation and parsers is currently too hard coded making it difficult to adapt to varied user requirements.

Also, input directives own when the viewValue is updated (via a call to $setViewValue). This makes it difficult to enable ngModelOptions to declare when updates occur, such as debouncing or alternative update events.

ngControl -> ngModel directive (shim over ngControl directive)

ngModel

[disabled]="someExpr"

(click)="handleClick"

([ngModel]) =

<!-- angular 1.5.x -->  
$scope.form = new Form();

$scope.form.valueInput = new Control(...);  
$scope.form.control.required = true;

<input ng-control=”valueInput”>

## <!-- angular 1.5x with angular-model.js --> <input ng-model=”value” required>

## 

## Overview

This design improves the API of the current components, such as NgModelControllerasyn, so there is less coupling between them. This makes it possible to connect them together in different ways to provide more varied behaviour.



### InputController

### This design defines a new abstraction called InputController that clarifies the interface to the input directive and decouples it from the NgModelController and ngModelOptions settings.

### NgModelController and FormController

The programmatic interfaces to **NgModelController** and **FormController** are simplified with more points to hook into the processing of validation and transformation. It will be possible to configure the **NgModelController** and **FormController** to behave in user defined ways via **Adaptors**.

### Adaptors

The **ngModelOptions** directive already allows the application developer to modify the behaviour of **ngModel**, **form** and **input directives** at specific points in the DOM. The new design extends **ngModelOptions** to allow the application developer to specify "**Adaptors**". **Adaptors** are injectable components that will be called with instances of relevant components, (such as **NgModelController**, **FormController**) so they can be configured and connected.

## Proof of Concept

## There is a basic Proof of Concept (POC) in GitHub: <https://github.com/petebacondarwin/ngModelPOC>.

# Design Details

## InputController

### Problems

In the original design, **input directives** control when an update to the **$viewValue** occurs by calling **$setViewValue(newValue)**. In Angular 1.3, we introduced the idea of letting the application developer specify what events triggered an update to the **$viewValue**, and also to specify debounce delay for these events.

This new feature was at odds with the original design, since we really needed to change what controlled when the **$viewValue** was updated: from hard coded DOM event handlers in the **input directives** to arbitrary events specified in **ngModelOptions**. Since we didn't want to break the current functionality, we created a complicated and unwieldy system of extra methods, such as **$commitViewValue**, **$rollBackViewValue** and **$$debounceViewValueCommit** to manage this interaction. This made it more difficult to understand how ngModel worked, conceptually, and also led to numerous problematic corner cases and issues.

### Solutions

To allow arbitrary events to trigger when the **$viewValue** is updated, we need to redesign the interaction between the **input directive**, **ngModelOptions** and the **NgModelController**.

Rather than expecting the **input directive** to "push" the current value to the **NgModelController**, the new design inverts this control so that the **NgModelController** can "pull" the current value from the **input directive** when it needs it. This makes it much simpler for us to specify when to trigger an update to the **$viewValue**, via **ngModelOptions**.

To support this the new design introduces a concept called an **InputController**. This is a component that is used by the **NgModelController** to talk to the **input directive** (and its underlying DOM elements).

The **InputController** does two things:

* **Read/Write access to the input value**: This decouples the underlying DOM implementation of the input control (e.g. an input element, a select element, or a complex directive that generates its own custom HTML, such as a date-picker or a WebComponent) from the value that the input control represents.
* **Mapping of DOM events to abstract input events**: This decouples the DOM events that are specific to the **input directive**, or are specified via **ngModelOptions**, from the abstract **Input Events** that are handled by the **NgModelController** or other components.

**Input directives** are responsible for creating an InputController and providing it to **NgModelController**. If an **input directive** does not provide an **InputController** (as in legacy cases) then the **ngModel** directive will create a default/legacy **InputController** instance and use that instead.

### Accessing the Input Value

### For reading and writing the value of an input directive, the InputController exposes $readValue() and $writeValue(value) methods that should be customized by the associated input directive. The agreement is that NgModelController will call $readValue() when it believes that the input has changed (perhaps because of a "change" input event) and $writeValue(value) when it wants to update the input control's value (such as when the $viewValue has been updated programmatically).

### Also, the design moves the responsibility for deciding whether the input is "empty" into the InputController by providing the $isEmpty() method.

To support the many legacy input directives that will not provide their own **InputController**, the **ngModel** directive will create and use a special legacy **InputController** instance. This legacy **InputController**, will contain much of the current code that deals with caching calls to **$setViewValue()** so that the **NgModelController** can get the correct value when it calls **$readValue()**. Over time it is hoped that these legacy **input directives** will migrate to support the **InputController** concept so that we can deprecate and remove the legacy **InputController**.

### Input Event Mapping

The **NgModelController** should not have to know about what concrete DOM events trigger changes to its state. Instead it should only be interested in a set of abstract **Input Events**, such as "changed" or "touched". The **InputController** holds a mapping between real **DOM events** and abstract **Input Events**. These mappings are configured via the **ngModelOptions** settings.

The following diagram illustrates an event mapping in an **InputController**:



The mapping of events provides a complete decoupling of the **DOM events** from the **NgModelController**. The mapping of **DOM events** to **input events** is a many-to-many relationship. This means that a **DOM event** (such as "blur" for example) can trigger multiple **input events** (such as "changed" and "touched") and also that many **DOM events** (such as "keydown" and "input") can trigger a single **Input Event** (such as "changed").

The **input directive** itself can provide a "default" set of event mappings, in the **InputController.$defaultMappings()**. These mappings are specific to the functioning of that directive, these can be added to the **InputController**'s event mappings. Other mappings may be provided *externally* via an **ngModelOptions** directive. Event mappings can also specify **debounce delays** so that the **NgModelController** will know that when it receives an **input event** it has been suitably debounced.

The agreement is that the **NgModelController** listens to the "changed" **input event**. The legacy InputController will provide a default mapping for the "changed" **input event** that is triggered by calls to **$setViewValue()**, as is currently implemented in the **NgModelController**.

### InputController API

The public API for the **InputController** is:

|  |
| --- |
| $readValue() : \*  $writeValue(value : \*)  $isEmpty() : Boolean  $inputEvents : Array<EventList>  $defaultEventMappings : EventMapping[]  $applyEventMappings(mappings : EventMapping[]) |

## NgModelController

The NgModelController is responsible for converting models from view values to model values and vice versa; it is also responsible for exposing the validity of the model and view values.

### Problems

Much of the workings of **ngModel** is hard coded into the **NgModelController** or the **input directives** themselves. This makes it difficult to adapt how the components work to provide different behaviour (such as changing when and how the scope is updated based on validity) or to improve performance (for instance not updating state information, such as **$touched**, if not needed).

The **$modelValue** and **$viewValue**, along with validation and transformation were originally designed with simple values (such as strings and numbers) in mind. They do not provide good support for values that are collections or complex objects or even **undefined**.

The **$parsers** and **$formatters** are not able to specify the reasons for a processing failure and it is not possible to provide additional context to validation errors. Validation state also relies on limited non-intuitive *magic* values, **null** (for validation skipped) and **undefined** (for validation pending).

Finally, there is little support in **ngModel** for more sophisticated input directives, that can format their display on the fly (such as masked inputs), validate between related inputs and grouping radio buttons.

### Solutions

## Rather than providing a built in pipeline for processing, the new design of NgModelController exposes decoupled methods, properties and events that can be wired up using adaptors that are appropriate for a particular application (or part of an application).

### Transforms

Each **NgModelController** instance contains a **$parsers** and a **$formatters** property, which were previously arrays of simple functions that take a value and return a value. This design introduces a new abstraction **Transform**, which can contain additional useful information about these transformation functions, such as a name and whether they expect to be passed a collection or not. A **parser** and a **formatter** are just instances of a **Transform**.

For backward compatibility plain **parser** or **formatter** functions will be automatically wrapped into a **Transform** object during post link of the **ngModel** directive.

#### Triggering Transformation

**Transform** functions are still synchronous operations. More complex asynchronous transformation would have to be done on the $scope level. The **$parsers** are triggered by calling **$parseView()** and the **$formatters** are triggered by calling **$formatModel()**.

The **NgModelController** exposes the **$viewParsed** and **$modelFormatted** events that are triggered when a call to **$parseView()** or **$formatModel()**, respectively, have completed successfully. They can be handled to do further processing (such as validation and writing to the scope or **InputController)**.

#### Transform Errors

If a **Transform** is unable to parse or format a value then it should throw an error, containing relevant information. This stops the processing of the pipeline and throws a **TransformError**. Previously **$parsers** and **$formatters** just returned **null** or **undefined** if they were not valid.

For backward compatibility, legacy **$parsers** and **$formatters** are wrapped in a special instance of a **Transform** that will throw an error if the **$parser** or **$formatter** returns **null** or **undefined**.

#### Array Handling

A **Transform** can specify that it expects a collection rather than a single value to be passed to it by setting its **expectArray** property to **true**.

When a collection is passed to a **Transform** that expects collections or a non-collection is passed to a **Transform** that does not expect a collection, the **Transform function** simply receives the value as-is.

If a collection is passed to a **Transform** that does not expect a collection then the **Transform function** receives each of the items in the collection one at a time, along with the index of the item in the collection and the whole collection as additional parameters.

When a non-collection is passed to a **Transform** that expects collections, the value is wrapped as a single item array that is passed to the **Transform function**.

### Validations

Validity of a value is determined by passing the value to **Validators**. **Validators** can be synchronous, returning their **Validation** result immediately, or they can be asynchronous, returning a Promise to their **Validation** result.

Each **NgModelController** instance contains two properties **$validators** and **$asyncValidators**. Originally, each item in these arrays was a simple function that takes a value and returns either a **Boolean**, for synchronous validation, or a **Promise** for a **Boolean**, for asynchronous validation. This design introduces a new abstraction called **Validator**, which holds the **Validator function** and additional information such as whether the **Validator** expects to be passed a collection or not.

Previously, **Validators** were simple functions that can take a **modelValue** and a **viewValue** as parameters. It turns out that most validators don't care for the difference; in the few cases where they do use both, they almost always just validate the first truthy value out of the **modelValue** and **viewValue**. In this design, we drop this distinction and just validate a single value, which is likely to be the **viewValue**.

#### Return Value

**Validator functions** can return one of three types of values:

* **Boolean**: true for valid, false for invalid
* **Validation**: an object containing a **$isValid** property for the validity and any custom error information.
* **Promise<boolean|Validation>**: a **Promise** to a **Boolean** or **Validation** object, for when the validator is asynchronous.

**Validators** can provide more contextual information about the error by adding extra properties to the **Validation** object that they return.

#### Array Handling

A **Validator** can specify that it expects a collection rather than a single value to be passed to it by setting its **expectArray** property to **true**.

When a collection is passed to a **Validator** that expects collections or a non-collection is passed to a **Validator** that does not expect a collection, the **Validator function** simply receives the value as-is.

If a collection is passed to a **Validator** that does not expect a collection then the **Validator function** receives each of the items in the collection one at a time, along with the index of the item in the collection and the whole collection as additional parameters.

When a non-collection is passed to a **Validator** that expects collections, the value is wrapped as a single item array that is passed to the **Validator function**.

#### Triggering Validation

You trigger a validation run by calling **NgModelController.$validate(value)**. This returns a Promise to a set of **ValidationResults**. The **$isValid** property of this is true if all the validations resolved to valid and false if *at least one* validation resolved to invalid. You can then inspect the validations property for more information.

**Validators** in the **$validators** array are run synchronously. If all these synchronous **Validators** report that the value is valid, then the **$asyncValidators** are called. If any of the synchronous Validators report that the value is invalid then the **$asyncValidators** are not run.

The **ValidationResults** **Promise** will resolve *as soon as any of the* ***Validators*** *resolves to invalid*. This speeds up validation and means that the client code doesn't have to wait for all the, potentially slow, async **Validators** to complete.

For backward compatibility, a call to **$validate(value)** will run **Validators** found in both the **$validators** and **$asyncValidators** properties.

### State Management

The NgModelController holds states, which are exposed on the $scope as its properties and on the DOM element, as CSS classes.

### NgModelController API

The public API for the NgModelController is:

## Adaptors

### Problems

The **NgModelController** and related components are hard coded to a specific behaviour, which is not always appropriate for all application scenarios. An example of this is when to update the scope: there are valid arguments for both updating and not updating the model value on the scope when the value is determined to be invalid.

Since it appears that there is no single acceptable solution to these use cases, we need a mechanism to allow different applications (and parts of applications) to vary the behaviour from the default provided by AngularJS.

### Solutions

## This design capitalizes on the ngModelOptions directive's ability to specify settings at different points in the DOM, which apply to all the elements below the directive. We will allow the ngModelOptions directive to specify an Adaptor for each of the main ngModel related controllers: NgModelController and FormController.

#### Registering Adaptors

Adaptors are registered in the **$modelOptions** service by calling **$modelOptions.registerAdaptor**. The adaptor function will take a single parameter, which is the component to adapt.

|  |
| --- |
| $modelOptions.registerAdaptor('modelAdaptor', function(ngModelController) {  // Configure the controller  }); |

#### Applying Adaptors

**Adaptors** can be applied to components by specifying them in the **adaptors** property of **ngModelOptions** - the property key is the name of the component to adapt and the value is the name of an adaptor registered with **$modelOptions**.

An appropriate adaptor is found by searching up the **ngModelOptions** directives to find the first instance that specifies an adaptor for the given component.

For example, consider the following configuration:

|  |
| --- |
| <body ng-model-options=  "{adaptors:{ngModel:'modelAdaptor', form:'formAdaptor'}}">  <form ng-model-options="{adaptors:{form:'customFormAdaptor'}}">  <input ng-model="xx">  <input ng-model="yy"  ng-model-options="{adaptors:{ngModel:'customModelAdaptor'}}">  </form>  </body> |

Default adaptors for **ngModel** and **form** are specified on the **<body>** element, a custom adaptor for **form** is specified on the **<form>** element and there is a custom model adaptor specified on one of the **<input>** elements.

The **FormController** attached to the **<form>** element will be adapted by the adaptor registered as **customFormAdaptor**. The **NgModelController** attached to **xx** will use the adaptor registered as **modelAdaptor** while the **NgModelController** attached to **yy** will use the adaptor registered as **customModelAdaptor**.

# What this design doesn't try to handle …

Stuff that will be fixed in Angular 2!

# Overview of Problems

This section contains details of the most significant pain points in the current implementation. The highlights are:

1. Models/Views get changed based on validation state
2. Custom parsers are not able to specify a name
3. Parsers/formatters cannot return `undefined` as a valid value
4. Parsers/formatters cannot specify the reason for a processing failure
5. Models/Views that are collections are not well supported
6. Validation between related controls is complex to implement
7. Setting the state of a control is not flexible or performant enough
8. Radio control groups are not well supported
9. Formatted inputs (such as masked inputs) are not well supported
10. It is not possible to provide additional context to validation errors
11. Validation state relies on limited unintuitive *magic* values, `null` (skipped) and `undefined` (pending)

## Models/Views get changed based on validation state

By default, we set the scope model to undefined if a validator finds an error.

This behavior can be turned off by setting ngModelOptions.allowInvalid to true, but this is not enough for the following cases:

* someone always wants to keep a valid modelValue instead of an invalid modelValue (see [Bug: $asyncValidators destroys data bound model property](https://github.com/angular/angular.js/issues/10035))
* you want to reset a model after it was set programmatically. We changed that in 1.3.0, because it is definitely the saner default. Still, making the “modelBound” callback available seems like a good idea. (see [$validators should reset model to undefined upon invalid initial value](https://github.com/angular/angular.js/issues/10286) and [No simple way to listen for the NgModelController loading](https://github.com/angular/angular.js/issues/5042))
* people expect that setting a model to undefined will also reset the view. However, this does not happen if an invalid value has been entered into the view, and the model has never been changed from undefined ([Input[type=email] is not reset on model reset](https://github.com/angular/angular.js/issues/10027))

### Ideas:

* Instead of putting more options into ngModelOptions, we could provide an api called $updateModel, that is similar to $render, only in the other direction. It would receive / have access to modelValue, validity, prevModelValue and prevValidity and maybe origin of the action (programmatic, viewValue update (user input), $validate). Based on this the user could decide if the model should be updated and with what value.
* introduce listeners similar to ngChange that can call the modelUpdate function based on the ngModel state. Add “default” listeners that handle updating / resetting models based on our current specs, but allow easy overriding / customization.

## Parse Error API is flawed (needs breaking):

The current design is flawed in a two ways:

1. Built-in parsers define their name (which is used as the error property) via ctrl.$$parserName. Custom parsers can currently not do this, and use 'parse' as error. When you have a built in parser that passes, and a custom parser that does not, the error set will nevertheless be that of ctrl.$$parserName, since that has precedence and is set in the linking phase. So far, there has been no bug report about this, but it's still less than ideal.

2. Since returning undefined indicated a parse error, a user cannot deliberately set undefined as a return value. This is problematic when you use a control as a filter variable, see <https://github.com/angular/angular.js/issues/10230>

3. returning undefined means there is no way for a parser to indicate \*what\* the problem was. e.g parsing CSV values, we could have numerous reasons for a failed parse: no delimiters, unterminated quotes, etc.

### Ideas:

For 1, Simple idea is to allow setting the parser name when adding the parser, possibly by making the argument an array, e.g. $parsers.push(['customParser', func]). This is quite ugly, though and required to analyse the parser when calling it.

A better idea might be to introduce an explicit $setParseError('parserName') function that must be called from inside a parser on parse error - the return value could then be anything (or would default to undefined). This could also provide the error context for (3).

Another idea is to have the parser throw an error if it fails.

## Collection Model Values don’t work very well:

* Changes inside collections are not picked up because we use === for detecting changes
* Validators cannot validate collections of values (not even arrays of strings)
* ngList is already crippled by this
* there is not support for multiple viewValues. HTML5 allows this for emails, but we could easily extend it to other text based inputs
* replace ngList with multiple? See [ngList is mistaken for the list attribute](https://github.com/angular/angular.js/issues/9132)

### Related issues:

* [Dynamic input[radio] doesn't remain checked with complex dynamic data](https://github.com/angular/angular.js/issues/8014)
* [ngModelController: trigger change when only property of model changes](https://github.com/angular/angular.js/issues/5449)
* [ngModel + ngList binding to array does not update view when model changes](https://github.com/angular/angular.js/issues/1751)

### Problems / Status:

* Before we can do this, we need better support for collection model values
* validation pipeline lacks support for validating arrays
* ngModelController currently stores the viewValue as string (unsuitable for validating multiple viewValues, e.g. min/maxlength)
* parsers start with the viewValue (string). Should parsers be made multiple-aware, i.e. each element extracted from multiple is passed through the parser chain?
* shahata already proposed a PR for multiple / ngList, which adds a listParser that splits the control value into an array and allows the validation engine to validate the single elements. (<https://github.com/angular/angular.js/pull/8987> )
* Parsers would also only receive the list array. That means custom parsers need to be written with multiple in mind. I think this is acceptable, as the number of controls that heavily make use of parsers and allow multiple will be low (hopefully). We'd need to update our internal parsers for this, though.

## The control states (dirty, touched etc.) are never enough for everyone’s needs

There are many different ways of displaying validation errors, and people have requested different additions so they can support their favorite out of the box:

* a **focused** state (<https://github.com/angular/angular.js/pull/10215>)
* a **hasViewValue** state
* a **hasBeenSetFromScope** state <https://github.com/angular/angular.js/issues/10050>
* first validation display on blur, subsequent validation display immediately <https://github.com/angular/angular.js/issues/10108>
* [**$dirtyAfter state**](https://github.com/angular/angular.js/pull/5888)(allows to see if a input has been skipped without entering anything)
* **resetting the touched state** when the user re-focusses an input

### Ideas

The problem with adding these features to core is that every new event listener or class set on the control will slow down the application (we already had a performance regression for ngModelController in 1.3)

Instead of adding everyone’s favorite state, see if it is possible / worth the effort to make it really easy to add new states by hooking into ngModel life cycle events. (<https://github.com/angular/angular.js/issues/5042>), view updated, model updated -> viewChangeListeners). This would also clean up parsers and formatters, which are currently abused for a lot of these things.

Karl Seamon mentioned that doubleclick has the concept of warnings and alerts additional to errors

* make digest heavy states optional (blur, focus)

## Problems with Radio Inputs

* each radio input in a group has an individual controller, but since they are grouped by name, the last registered radio is registered / published as a controller on a form ([Multiple inputs with the same name bug](https://github.com/angular/angular.js/issues/7647) ) -> ideally, a group radio should have one controller
* Radios in separate scopes are not bound to same model <https://github.com/angular/angular.js/issues/10043>
* Radio button extends outside form: <https://github.com/angular/angular.js/issues/8506> -> normal html
* Radios without a name are not grouped together if they have the same model

### Ideas

Could this be solved simply by providing a new input directive (say ngRadioGroup) that allows us to group radio buttons together, maybe allowing a template to be provided similar to how ngMessages now works?

## Hard to create fields that automatically format the <input>

Eg. Phone numbers (that add the parentheses /spaces automatically)

Numbers that follow the locale format

### Problem:

We provide no API to manipulate the viewValue after it has been input, but before it is parsed.

Since there's no API, people have been using parsers to format the view / input before it is parsed

further / validated:

Most people use something like this inside a parser to format a viewValue on the fly:

//some condition

ngModelCtrl.$setViewValue(formattedVal);

ngModelCtrl.$commitViewValue(); // needed when the default commit is debounced / uses custom triggers

ngModelCtrl.$render();

This is brittle -- we had to fix two bugs that were introduced in 1.3 that made this difficult in some circumstances.

It also abuses the parsers: parsers should only be used for transforming the viewValue

### Ideas:

introduce a new API that allows to manipulate the viewValue and the control display value, before it is passed to the parsers.

This could work similar to the formatters pipeline.

It also makes sense to make it configurable when the view is updated, similar to ngModelOptions.

For example, if I type a date as 20141210, it makes sense not to interrupt my typing by inserting separators - this should happen during the blur event.

We'd probably also have to provide an API for setting the cursor in the input. Afaik, this can become pretty messy, especially for contenteditable fields. So it might be a good idea to make this pluggable.

* $setViewValue could also be overloaded to do this - encourage its modification?

## Hard to do validation that depends on other fields.

Eg. A change password field where the form contains `New Password` and `Confirm New Password`. It should only be valid if both match

### Problem:

it's not possible for an ngModel to require a sibling controller. For the model value, you can work around this by

watching the second model.

If both ngModels are part of a form, it should be much simpler - you can match on the $modelValue

but there's still the the problem that the $modelValue will change to undefined if the model is invalid. That means when matchModel is already invalid, and you change masterModel so that both control values match, you will always compare masterModel to the undefined matchModel (see <http://plnkr.co/edit/c1kDeK1pGoPqVjgmVEfP?p=preview>)

### Ideas:

Personally, I think having this functionality working inside a form context should be enough; also this might work very well as a curated 3rd party plugin. The linked plunker should be able to achieve this when we apply the current semantics of $$rawModelValue to $modelValue, that means the $modelValue always reflects the current state of the model value - either set from the scope, or the parsed result. Specifically, we would never set it to undefined when the validation failed.

Suggested API might be:

<input ng-model="password" ng-compare-with="passwordConfirmation" ng-compare-isvalid="password == $other">

## Validators can’t return structured errors

The `$error` object ends up with boolean flags. It’d be good to be able to return an object from validators that provides more information about the error, which the view can then use to provide localized error messages.

## setValidity’s API is unintuitive

Having 4 primitive values to indicate different states is unintuitive, and would normally require comments in application code. Allowing $setValidity(“x”, “pending”) or “skipped” would be more explicit.

# “Missing” features

## “form” attribute on inputs

* see <https://html.spec.whatwg.org/#association-of-controls-and-forms>

### Problems:

Getting DOM elements by name completely circumvents the scope hierarchy. Do we want this?

## Independent sub-forms:

Sub forms that have no effect on the validity of the parent form.

Proposed PR: <https://github.com/angular/angular.js/pull/10193>

### Problems:

ng-form is not meant as a complete replacement for form; notably submit handling from buttons, keypresses etc. is missing. We’d need to decide if we want to emulate this.

# Thoughts / concepts…

* Parsers : View -> Model
* Formatters : Model -> View
* View validators : only check the view
* Model validators : only check the model
* State :
  + valid/invalid/pending
  + pristine/dirty
  + untouched/touched
* Should the state affect the model/view values - if so in what way?
  + does state affect only view->model changes or both directions?
  + model/view updated anyway?
  + model/view set to specific value (null/undefined)?
  + model/view is left alone until state is valid
* Committed ViewValue
* viewChangeListeners:
  + contrary to their name, they are only called if the model changes (except on init).
  + I’d like to explore adding more of these hooks to make it easier for people to customize their stuff (e.g. **validityChangeListeners**, an actual **viewChangeListener**, and a **modelBoundListener**)
* Ignore Empty Nested Form validation

# Performance:

every event handler, (built-in and users') calls its own apply (Karl Seamon) -> combine multiple event handler callbacks

keep digest to isolate scope (Karl Seamon) (This is a more general issue)

# Useful Contacts

* Karl Seamon
* Ward Bell
* Matias
* Misko
* Victor / Rado

# Meetings

## 2014/12/16

* a rough design / vision is needed before we can talk about single issues
* clarification needed how much we want to break the API (possible grievances for the community)
* don’t put more stuff into ngModelOptions
* evaluate if it makes sense to align the refactoring with angular 2 forms design (<https://docs.google.com/document/d/1US9h0ORqBltl71TlEU6s76ix8SUnOLE2jabHVg9xxEA/edit#heading=h.s3llc14qem13> )

## Matias some ideas about Server-side validation / ngMessages

* evaluate if it makes sense to break up ngModel into smaller parts
* validation pipeline could be an optional part
* e.g. validation is not needed for text inputs that are used as filters

# Caveats

## Backward Compatibility!!

# Security Considerations

How you’ll be secure

# Performance Considerations / Test Strategy

We need to put in place a bunch of BenchPress benchmarks to ensure that these changes do not impact negatively on performance.

# Work Breakdown

Description of development phases and approximate time estimates.

# 