# BUILD YOUR OWN ANGULARJS

  

# Build Your Own AngularJS

Tero Parviainen

ISBN 978-952-93-3544-2

# Introduction

This book is written for the working programmer, who either wants to learn AngularJS, or already knows AngularJS and wants to know what makes it tick.

AngularJS is not a small framework. It has a large surface area with many new concepts to grasp. Its codebase is also substantial, with 35K lines of JavaScript in it. While all of those new concepts and all of those lines of code give you powerful tools to build the apps you need, they also come with a learning curve.

I hate working with technologies I don't quite understand. Too often, it leads to code that just happens to work, not because you truly understand what it does, but because you went through a lot of trial and error to make it work. Code like that is difficult to change and debug. You can't reason your way through problems. You just poke at the code until it all seems to align.

Frameworks like AngularJS, powerful as they are, are prone to this kind of code. Do you understand how Angular does dependency injection? Do you know the mechanics of scope inheritance? What exactly happens during directive transclusion? When you don't know how these things work, as I didn't when I started working with Angular, you just have to go by what the documentation says. When that isn't enough, you try different things until you have what you need.

The thing is, while there's a lot of code in AngularJS, it's all just code. It's no different from the code in your applications. Most of it is well-factored, readable code. You can study it to learn how Angular does what it does. When you've done that, you're much better equipped to deal with the issues you face in your daily application development work. You'll know not only what features Angular provides to solve a particular problem, but also how those features work, how to get the most out of them, and where they fall short.

The purpose of this book is to help you demystify the inner workings of AngularJS. To take it apart and put it back together again, in order to truly understand how it works.

A true craftsman knows their tools well. So well that they could in fact make their own tools if needed. This book will help you get there with AngularJS.

## How To Read This Book

During the course of the book we will be building an implementation of AngularJS. We'll start from the very beginning, and in each chapter extend the implementation with new capabilities.

While there are certain areas of functionality in Angular that are largely independent, most of the code you'll be writing builds on things implemented in previous chapters. That is why a sequential reading will help you get the most out of this book.

The format of the book is simple: Each feature is introduced by discussing what it is and why it's needed. We will then proceed to implement the feature following test-driven development practices: By writing failing tests and then writing the code to make them pass. As a result, we will produce not only the framework code, but also a test suite for it.

It is highly encouraged that you not only read the code, but also actually type it in and build your own Angular while reading the book. To really make sure you've grasped a concept, poke it from different directions: Write additional test cases. Try to intentionally break it in a few ways. Refactor the code to match your own style while keeping the tests passing.

If you're only interested in certain parts of the framework, feel free to skip to the chapters that interest you. While you may need to reference back occasionally, you should be able to poach the best bits of Angular to your own application or framework with little difficulty.

## Source Code

The source code and test suite implemented in this book can be found on GitHub, at https://github.com/teropa/build-your-own-angularjs/.

To make following along easier, commits in the repository are ordered to match the order of events in the book. Each commit message references the corresponding section title in the book. Note that this means that during the production of the book, the history of the code repository may change as revisions are made.

There is also a Git tag for each chapter, pointing to the state of the codebase at the end of that chapter. You can download archives of the code corresponding to these tags from https://github.com/teropa/build-your-own-angularjs/releases.

## Contributors

I would like to thank the following people for their valuable feedback and help during the writing of this book:

- Iftach Bar
- Xi Chen
- Wil Pannell
- Pavel Pomyerantsyev
- Mauricio Poppe
- Mika Ristimäki
- Jesus Rodriguez
- Scott Silvi

## Early Access: Errata & Contributing

This book is under active development, and you are reading an early access version. The content is still incomplete, and is likely to contain bugs, typos, and other errors.

As an early access subscriber, you will receive updated versions of the book throughout its production process.

Your feedback is more than welcome. If you come across errors, or just feel like something could be improved, please file an issue to the book's Errata on GitHub: https://github.com/teropa/build-your-own-angularjs/issues. To make this process easier, there are links in the footer of each page that take you directly to the corresponding chapter's errata, and to the form with which you can file an issue.

## Contact

Feel free to get in touch with me by sending an email to tero@teropa.info or tweeting at @teropa.

  

  

# Part I

# Scopes

We will begin our implementation of AngularJS with one of its central building blocks: Scopes. Scopes are used for many different purposes:

- Sharing data between controllers and views
- Sharing data between different parts of the application
- Broadcasting and listening for events
- Watching for changes in data

Of these several use cases, the last one is arguably the most interesting one. Angular scopes implement a *dirty-checking* mechanism, using which you can get notified when a piece of data on a scope changes. It can be used as-is, but it is also the secret sauce of *data binding*, one of Angular's primary selling points.

In this first part of the book you will implement Angular scopes. We will cover four main areas of functionality:

1. The digest cycle and dirty-checking itself, including `$watch`, `$digest`, and `$apply`.
2. Scope inheritance – the mechanism that makes it possible to create scope hierarchies for sharing data and events.
3. Efficient dirty-checking for collections – arrays and objects.
4. The event system – `$on`, `$emit`, and `$broadcast`.

  

  

# Chapter 1

# Scopes And Digest

Angular scopes are plain old JavaScript objects, on which you can attach properties just like you would on any other object. However, they also have some added capabilities for observing changes in data structures. These observation capabilities are implemented using *dirty-checking* and executed in a *digest cycle*. That is what we will implement in this chapter.

## Scope Objects

Scopes are created by using the `new` operator on a `Scope` constructor. The result is a plain old JavaScript object. Let's make our very first test case for this basic behavior.

Create a test file for scopes in `test/scope_spec.js` and add the following test case to it:

*test/scope_spec.js*

```
'use strict';

var Scope = require('../src/scope');

describe("Scope", function() {

  it("can be constructed and used as an object", function() {
    var scope = new Scope();
    scope.aProperty = 1;

    expect(scope.aProperty).toBe(1);
  });

});
```

On the top of the file we enable ES5 strict mode, and require `Scope`, which we are expecting to find under the `src` directory.

This test just creates a `Scope`, assigns an arbitrary property on it and checks that it was indeed assigned.

> If you have Karma running in a terminal, you will see it fail after you've added this test case, because we haven't implemented `Scope` yet. This is exactly what we want, since an important step in test-driven development is seeing the test fail first. Throughout the book I'll assume the test suite is being continuously executed, and will not explicitly mention when tests should be run.

We can make this test pass easily enough: Create `src/scope.js` and set the contents as:

*src/scope.js*

```
'use strict';

function Scope() {

}

module.exports = Scope;
```

In the test case we're assigning a property (`aProperty`) on the scope. This is exactly how properties on the Scope work. They are plain JavaScript properties and there's nothing special about them. There are no special setters you need to call, nor restrictions on what values you assign. Where the magic happens instead is in two very special functions: `$watch` and `$digest`. Let's turn our attention to them.

## Watching Object Properties: $watch And $digest

`$watch` and `$digest` are two sides of the same coin. Together they form the core of what the digest cycle is all about: Reacting to changes in data.

With `$watch` you can attach a *watcher* to a scope. A watcher is something that is notified when a change occurs in the scope. You create a watcher by providing two functions to `$watch`:

- A *watch function*, which specifies the piece of data you're interested in.
- A *listener function* which will be called whenever that data changes.

  

> As an Angular user, you actually usually specify a watch *expression* instead of a watch function. A watch expression is a string, like `"user.firstName"`, that you specify in a data binding, a directive attribute, or in JavaScript code. It is parsed and compiled into a watch function by Angular internally. We will implement this in Part 2 of the book. Until then we'll use the slightly lower-level approach of providing watch functions directly.

The other side of the coin is the `$digest` function. It iterates over all the watchers that have been attached on the scope, and runs their watch and listener functions accordingly.

To flesh out these building blocks, let's define a test case which asserts that you can register a watcher using `$watch`, and that the watcher's listener function is invoked when someone calls `$digest`.

To make things a bit easier to manage, add the test to a nested `describe` block in `scope_spec.js`. Also create a `beforeEach` function that initializes the scope, so that we won't have to repeat it for each test:

*test/scope_spec.js*

```
describe("Scope", function() {

  it("can be constructed and used as an object", function() {
    var scope = new Scope();
    scope.aProperty = 1;

    expect(scope.aProperty).toBe(1);
  });

  describe("digest", function() {

    var scope;

    beforeEach(function() {
      scope = new Scope();
    });

    it("calls the listener function of a watch on first $digest", function() {
      var watchFn    = function() { return 'wat'; };
      var listenerFn = jasmine.createSpy();
      scope.$watch(watchFn, listenerFn);

      scope.$digest();

      expect(listenerFn).toHaveBeenCalled();
    });

  });

});
```

  

In the test case we invoke `$watch` to register a watcher on the scope. We're not interested in the watch function just yet, so we just provide one that returns a constant value. As the listener function, we provide a Jasmine Spy. We then call `$digest` and check that the listener was indeed called.

> A spy is Jasmine terminology for a kind of mock function. It makes it convenient for us to answer questions like "Was this function called?" and "What arguments was it called with?"

There are a few things we need to do to make this test case pass. First of all, the Scope needs to have some place to store all the watchers that have been registered. Let's add an array for them in the `Scope` constructor:

*src/scope.js*

```
function Scope() {
  this.$$watchers = [];
}
```

> The double-dollar prefix `$$` signifies that this variable should be considered private to the Angular framework, and should not be called from application code.

Now we can define the `$watch` function. It'll take the two functions as arguments, and store them in the `$$watchers` array. We want every Scope object to have this function, so let's add it to the prototype of `Scope`:

*src/scope.js*

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn
  };
  this.$$watchers.push(watcher);
};
```

Finally there is the `$digest` function. For now, let's define a very simple version of it, which just iterates over all registered watchers and calls their listener functions:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  _.forEach(this.$$watchers, function(watcher) {
    watcher.listenerFn();
  });
};
```

  

This function is using the `forEach` function from LoDash, so we need to require LoDash at the top of the file:

*src/scope.js*

```
'use strict';

var _ = require('lodash');

// ...
```

The test passes but this version of `$digest` isn't very useful yet. What we really want is to check if the values specified by the watch functions have actually changed, and *only then* call the respective listener functions. This is called *dirty-checking*.

## Checking for Dirty Values

As described above, the watch function of a watcher should return the piece of data whose changes we are interested in. Usually that piece of data is something that exists on the scope. To make accessing the scope from the watch function more convenient, we want to call it with the current scope as an argument. A watch function that's interested in a `firstName` attribute on the scope may then do something like this:

```
function(scope) {
  return scope.firstName;
}
```

This is the general form that watch functions usually take: Pluck some value from the scope and return it.

Let's add a test case for checking that the scope is indeed provided as an argument to the watch function:

*test/scope_spec.js*

```
it("calls the watch function with the scope as the argument", function() {
  var watchFn    = jasmine.createSpy();
  var listenerFn = function() { };
  scope.$watch(watchFn, listenerFn);

  scope.$digest();

  expect(watchFn).toHaveBeenCalledWith(scope);
});
```

  

This time we create a Spy for the watch function and use it to check the watch invocation.

The simplest way to make this test pass is to modify `$digest` to do something like this:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var self = this;
  _.forEach(this.$$watchers, function(watcher) {
    watcher.watchFn(self);
    watcher.listenerFn();
  });
};
```

> The `var self = this;` pattern is something we'll be using throughout the book to get around JavaScript's peculiar binding of `this`. There is a good A List Apart article that describes the problem and the pattern.

Of course, this is not quite what we're after. The `$digest` function's job is really to call the watch function and compare its return value to whatever the same function returned last time. If the values differ, the watcher is *dirty* and its listener function should be called. Let's go ahead and add a test case for that:

*test/scope_spec.js*

```
it("calls the listener function when the watched value changes", function() {
  scope.someValue = 'a';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.someValue; },
    function(newValue, oldValue, scope) { scope.counter++; }
  );

  expect(scope.counter).toBe(0);

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.someValue = 'b';
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(2);

});
```

  

We first plop two attributes on the scope: A string and a number. We then attach a watcher that watches the string and increments the number when the string changes. The expectation is that the counter is incremented once during the first `$digest`, and then once every subsequent `$digest` *if* the value has changed.

Notice that we also specify the contract of the listener function: Just like the watch function, it takes the scope as an argument. It's also given the new and old values of the watcher. This makes it easier for application developers to check what exactly has changed.

To make this work, `$digest` has to remember what the last value of each watch function was. Since we already have an object for each watcher, we can conveniently store the last value there. Here's a new definition of `$digest` that checks for value changes for each watch function:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var self = this;
  var newValue, oldValue;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      watcher.last = newValue;
      watcher.listenerFn(newValue, oldValue, self);
    }
  });
};
```

For each watcher, we compare the return value of the watch function to what we've previously stored in the `last` attribute. If the values differ, we call the listener function, passing it both the new and old values, as well as the scope object itself. Finally, we set the `last` attribute of the watcher to the new return value, so we'll be able to compare to that next time.

We've now implemented the essence of Angular scopes: Attaching watches and running them in a digest.

We can also already see a couple of important performance characteristics that Angular scopes have:

- Attaching data to a scope does not by itself have an impact on performance. If no watcher is watching a property, it doesn't matter if it's on the scope or not. Angular does not iterate over the properties of a scope. It iterates over the watches.
- Every watch function is called during every `$digest`. For this reason, it's a good idea to pay attention to the number of watches you have, as well as the performance of each individual watch function or expression.

  

# Initializing Watch Values

Comparing a watch function's return value to the previous one stored in `last` works fine most of the time, but what does it do on the first time a watch is executed? Since we haven't set `last` at that point, it's going to be `undefined`. That doesn't quite work when the first *legitimate* value of the watch is also `undefined`. The listener should be invoked in this case as well, but it doesn't because our current implementation doesn't consider an initial `undefined` value as a "change":

*test/scope_spec.js*

```
it("calls listener when watch value is first undefined", function() {
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.someValue; },
    function(newValue, oldValue, scope) { scope.counter++; }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

We should be calling the listener function here too. What we need is to initialize the `last` attribute to something we can guarantee to be unique, so that it's different from anything a watch function might return, including `undefined`.

A *function* fits this purpose well, since JavaScript functions are so-called *reference values* - they are not considered equal to anything but themselves. Let's introduce a function value on the top level of `scope.js`:

*src/scope.js*

```
function initWatchVal() { }
```

Now we can stick this function into the `last` attribute of new watches:

*src/scope.js*

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
};
```

  

This way new watches will *always* have their listener functions invoked, whatever their watch functions might return.

What also happens though is the `initWatchVal` gets handed to listeners as the old value of the watch. We'd rather not leak that function outside of `scope.js`. For new watches, we should instead provide the new value as the old value:

*test/scope_spec.js*

```
it("calls listener with new value as old value the first time", function() {
  scope.someValue = 123;
  var oldValueGiven;

  scope.$watch(
    function(scope) { return scope.someValue; },
    function(newValue, oldValue, scope) { oldValueGiven = oldValue; }
  );

  scope.$digest();
  expect(oldValueGiven).toBe(123);
});
```

In `$digest`, as we call the listener, we just check if the old value is the initial value and replace it if so:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var self = this;
  var newValue, oldValue;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
    }
  });
};
```

# Getting Notified Of Digests

If you would like to be notified whenever an Angular scope is digested, you can make use of the fact that each watch is executed during each digest: Just register a watch without a listener function. Let's add a test case for this.

*test/scope_spec.js*

  

```
it("may have watchers that omit the listener function", function() {
  var watchFn = jasmine.createSpy().and.returnValue('something');
  scope.$watch(watchFn);

  scope.$digest();

  expect(watchFn).toHaveBeenCalled();
});
```

The watch doesn't necessarily have to return anything in a case like this, but it can, and in this case it does. When the scope is digested our current implementation throws an exception. That's because it's trying to invoke a non-existing listener function. To add support for this use case, we need to check if the listener is omitted in `$watch`, and if so, put an empty no-op function in its place:

*src/scope.js*

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
};
```

If you use this pattern, do keep in mind that Angular will look at the return value of `watchFn` even when there is no `listenerFn`. If you return a value, that value is subject to dirty-checking. To make sure your usage of this pattern doesn't cause extra work, just don't return anything. In that case the value of the watch will be constantly `undefined`.

## Keep Digesting While Dirty

The core of the implementation is now there, but we're still far from done. For instance, there's a fairly typical scenario we're not supporting yet: The listener functions themselves may also change properties on the scope. If this happens, and there's *another* watcher looking at the property that just changed, it might not notice the change during the same digest pass:

*test/scope_spec.js*

  

```
it("triggers chained watchers in the same digest", function() {
  scope.name = 'Jane';

  scope.$watch(
    function(scope) { return scope.nameUpper; },
    function(newValue, oldValue, scope) {
      if (newValue) {
        scope.initial = newValue.substring(0, 1) + '.';
      }
    }
  );

  scope.$watch(
    function(scope) { return scope.name; },
    function(newValue, oldValue, scope) {
      if (newValue) {
        scope.nameUpper = newValue.toUpperCase();
      }
    }
  );

  scope.$digest();
  expect(scope.initial).toBe('J.');

  scope.name = 'Bob';
  scope.$digest();
  expect(scope.initial).toBe('B.');

});
```

---

We have two watchers on this scope: One that watches the `nameUpper` property, and assigns `initial` based on that, and another that watches the `name` property and assigns `nameUpper` based on that. What we expect to happen is that when the `name` on the scope changes, the `nameUpper` and `initial` attributes are updated accordingly during the digest. This, however, is not the case.

> We're deliberately ordering the watches so that the dependent one is registered first. If the order was reversed, the test would pass right away because the watches would happen to be in just the right order. However, dependencies between watches do not rely on their registration order, as we're about to see.

What we need to do is to modify the digest so that it keeps iterating over all watches *until the watched values stop changing*. Doing multiple passes is the only way we can get changes applied for watchers that rely on other watchers.

First, let's rename our current `$digest` function to `$$digestOnce`, and adjust it so that it runs all the watchers once, and returns a boolean value that determines whether there were any changes or not:

*src/scope.js*

---

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    }
  });
  return dirty;
};
```

---

Then, let's redefine `$digest` so that it runs the "outer loop", calling `$$digestOnce` as long as changes keep occurring:

*src/scope.js*

---

```
Scope.prototype.$digest = function() {
  var dirty;
  do {
    dirty = this.$$digestOnce();
  } while (dirty);
};
```

---

`$digest` now runs all watchers *at least* once. If, on the first pass, any of the watched values has changed, the pass is marked dirty, and all watchers are run for a second time. This goes on until there's a full pass where none of the watched values has changed and the situation is deemed stable.

> Angular scopes don't actually have a function called `$$digestOnce`. Instead, the digest loops are all nested within `$digest`. Our goal is clarity over performance, so for our purposes it makes sense to extract the inner loop to a function.

We can now make another important observation about Angular watch functions: They may be run many times per each digest pass. This is why people often say watches should be *idempotent*: A watch function should have no side effects, or only side effects that can happen any number of times. If, for example, a watch function fires an Ajax request, there are no guarantees about how many requests your app is making.

  

## Giving Up On An Unstable Digest

In our current implementation there's one glaring omission: What happens if there are two watches looking at changes made by each other? That is, what if the state *never* stabilizes? Such a situation is shown by the test below:

*test/scope_spec.js*

```
it("gives up on the watches after 10 iterations", function() {
  scope.counterA = 0;
  scope.counterB = 0;

  scope.$watch(
    function(scope) { return scope.counterA; },
    function(newValue, oldValue, scope) {
      scope.counterB++;
    }
  );

  scope.$watch(
    function(scope) { return scope.counterB; },
    function(newValue, oldValue, scope) {
      scope.counterA++;
    }
  );

  expect((function() { scope.$digest(); })).toThrow();

});
```

We expect `scope.$digest` to throw an exception, but it never does. In fact, the test never finishes. That's because the two counters are dependent on each other, so on each iteration of `$$digestOnce` one of them is going to be dirty.

> Notice that we're not calling the `scope.$digest` function directly. Instead we're passing a function to Jasmine's `expect` function. It will call that function for us, so that it can check that it throws an exception like we expect.

> Since this test will never finish running you'll need to kill the Karma process and start it again once we've fixed the issue.

What we need to do is keep running the digest for some acceptable number of iterations. If the scope is still changing after those iterations we have to throw our hands up and declare it's probably never going to stabilize. At that point we might as well throw an exception, since whatever the state of the scope is it's unlikely to be what the user intended.

This maximum amount of iterations is called the TTL (short for "Time To Live"). By default it is set to 10. The number may seem small, but bear in mind this is a performance

  

sensitive area since digests happen often and each digest runs all watch functions. It's also unlikely that a user will have more than 10 watches chained back-to-back.

> It is actually possible to adjust the TTL in Angular. We will return to this later when we discuss providers and dependency injection.

Let's go ahead and add a loop counter to the outer digest loop. If it reaches the TTL, we'll throw an exception:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  do {
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

This updated version causes our interdependent watch example to throw an exception, as our test expected. This should keep the digest from running off on us.

## Short-Circuiting The Digest When The Last Watch Is Clean

In the current implementation, we keep iterating over the watch collection until we have witnessed one full round where every watch was clean (or where the TTL was reached).

Since there can be a large amount of watches in a digest loop, it is important to execute them as few times as possible. That is why we're going to apply one specific optimization to the digest loop.

Consider a situation with 100 watches on a scope. When we digest the scope, only the first of those 100 watches happens to be dirty. That single watch "dirties up" the whole digest round, and we have to do another round. On the second round, none of the watches are dirty and the digest ends. But we had to do 200 watch executions before we were done!

What we can do to cut the number of executions in half is to keep track of the last watch we have seen that was dirty. Then, whenever we encounter a clean watch, we check whether it's also the last watch we have seen that was dirty. If so, it means a full round has passed where no watch has been dirty. In that case there is no need to proceed to the end of the current round. We can exit immediately instead. Here's a test case for just that:

*test/scope_spec.js*

  

```
it("ends the digest when the last watch is clean", function() {

  scope.array = _.range(100);
  var watchExecutions = 0;

  _.times(100, function(i) {
    scope.$watch(
      function(scope) {
        watchExecutions++;
        return scope.array[i];
      },
      function(newValue, oldValue, scope) {
      }
    );
  });

  scope.$digest();
  expect(watchExecutions).toBe(200);

  scope.array[0] = 420;
  scope.$digest();
  expect(watchExecutions).toBe(301);

});
```

---

We first put an array of 100 items on the scope. We then attach a 100 watches, each watching a single item in the array. We also add a local variable that's incremented whenever a watch is run, so that we can keep track of the total number of watch executions.

We then run the digest once, just to initialize the watches. During that digest each watch is run twice.

Then we make a change to the very first item in the array. If the short-circuiting optimization were in effect, that would mean the digest would short-circuit on the first watch during second iteration and end immediately, making the number of total watch executions just 301 and not 400.

We don't yet have LoDash available in `scope_spec.js`, so we need to require it in in order to use the `range` and `times` functions:

*test/scope_spec.js*

```
'use strict';

var _ = require('lodash');
var Scope = require('../src/scope');
```

---

As mentioned, this optimization can be implemented by keeping track of the last dirty watch. Let's add a field for it to the `Scope` constructor:

*src/scope.js*

  

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
}
```

Now, whenever a digest begins, let's set this field to `null`:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

In `$$digestOnce`, whenever we encounter a dirty watch, let's assign it to this field:

*src/scope.js*

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      self.$$lastDirtyWatch = watcher;
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    }
  });
  return dirty;
};
```

Also in `$$digestOnce`, whenever we encounter a *clean* watch that also happens to have been the last dirty watch we saw, let's break out of the loop right away and return a falsy value to let the outer `$digest` loop know it should also stop iterating:

*src/scope.js*

  

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (newValue !== oldValue) {
      self.$$lastDirtyWatch = watcher;
      watcher.last = newValue;
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    } else if (self.$$lastDirtyWatch === watcher) {
      return false;
    }
  });
  return dirty;
};
```

Since we won't have seen any dirty watches this time, `dirty` will be `undefined`, and that'll be the return value of the function.

> Explicitly returning `false` in a `_.forEach` loop causes LoDash to short-circuit the loop and exit immmediately.

The optimization is now in effect. There's one corner case we need to cover though, which we can tease out by adding a watch from the listener of another watch:

*test/scope_spec.js*

```
it("does not end digest so that new watches are not run", function() {

  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.$watch(
        function(scope) { return scope.aValue; },
        function(newValue, oldValue, scope) {
          scope.counter++;
        }
      );
    }
  );
```

  

```
  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

---

The second watch is not being executed. The reason is that on the second digest iteration, just before the new watch would run, we're ending the digest because we're detecting the *first* watch as the last dirty watch that's now clean. Let's fix this by re-setting `$$lastDirtyWatch` when a watch is added, effectively disabling the optimization:

*src/scope.js*

---

```
Scope.prototype.$watch = function(watchFn, listenerFn) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
  this.$$lastDirtyWatch = null;
};
```

---

Now our digest cycle is potentially a lot faster than before. In a typical application, this optimization may not always eliminate iterations as effectively as in our example, but it does well enough on average that the Angular team has decided to include it.

Now, let's turn our attention to *how* we're actually detecting that something has changed.

# Value-Based Dirty-Checking

For now we've been comparing the old value to the new with the strict equality operator `===`. This is fine in most cases, as it detects changes to all primitives (numbers, strings, etc.) and also detects when an object or an array changes to a new one. But there is also another way Angular can detect changes, and that's detecting when something *inside* an object or an array changes. That is, you can watch for changes in *value*, not just in reference.

This kind of dirty-checking is activated by providing a third, optional boolean flag to the `$watch` function. When the flag is `true`, value-based checking is used. Let's add a test that expects this to be the case:

*test/scope_spec.js*

---

```
it("compares based on value if enabled", function() {
  scope.aValue = [1, 2, 3];
  scope.counter = 0;
```

  

```
  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    },
    true
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.aValue.push(4);
  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

---

The test increments a counter whenever the `scope.aValue` array changes. When we push an item to the array, we're expecting it to be noticed as a change, but it isn't. `scope.aValue` is still the same array, it just has different contents now.

Let's first redefine `$watch` to take the boolean flag and store it in the watcher:

*src/scope.js*

---

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    valueEq: !!valueEq,
    last: initWatchVal
  };
  this.$$watchers.push(watcher);
  this.$$lastDirtyWatch = null;
};
```

---

All we do is add the flag to the watcher, coercing it to a real boolean by negating it twice. When a user calls `$watch` without a third argument, `valueEq` will be `undefined`, which becomes `false` in the watcher object.

Value-based dirty-checking implies that if the old or new values are objects or arrays we have to iterate through everything contained in them. If there's any difference in the two values, the watcher is dirty. If the value has other objects or arrays nested within, those will also be recursively compared by value.

Angular ships with its own equal checking function, but we're going to use the one provided by Lo-Dash instead because it does everything we need at this point. Let's define a new function that takes two values and the boolean flag, and compares the values accordingly:

*src/scope.js*

---

  

```
Scope.prototype.$$areEqual = function(newValue, oldValue, valueEq) {
  if (valueEq) {
    return _.isEqual(newValue, oldValue);
  } else {
    return newValue === oldValue;
  }
};
```

---

In order to notice changes in value, we also need to change the way we store the old value for each watcher. It isn't enough to just store a reference to the current value, because any changes made within that value will also be applied to the reference we're holding. We would never notice any changes since essentially `$$areEqual` would always get two references to the same value. For this reason we need to make a deep copy of the value and store that instead.

Just like with the equality check, Angular ships with its own deep copying function, but for now we'll be using the one that comes with Lo-Dash.

Let's update `$digestOnce` so that it uses the new `$$areEqual` function, and also copies the `last` reference if needed:

*src/scope.js*

---

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    newValue = watcher.watchFn(self);
    oldValue = watcher.last;
    if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
      self.$$lastDirtyWatch = watcher;
      watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
      watcher.listenerFn(newValue,
        (oldValue === initWatchVal ? newValue : oldValue),
        self);
      dirty = true;
    } else if (self.$$lastDirtyWatch === watcher) {
      return false;
    }
  });
  return dirty;
};
```

---

Now our code supports both kinds of equality-checking, and our test passes.

Checking by value is obviously a more involved operation than just checking a reference. Sometimes a lot more involved. Walking a nested data structure takes time, and holding a deep copy of it also takes up memory. That's why Angular does not do value-based dirty checking by default. You need to explicitly set the flag to enable it.

  

> There's also a third dirty-checking mechanism Angular provides: Collection watching. We will implement it in Chapter 3.

Before we're done with value comparison, there's one more JavaScript quirk we need to handle.

## NaNs

In JavaScript, `NaN` (Not-a-Number) is not equal to itself. This may sound strange, and that's because it is. If we don't explicitly handle `NaN` in our dirty-checking function, a watch that has `NaN` as a value will always be dirty.

For value-based dirty-checking this case is already handled for us by the Lo-Dash `isEqual` function. For reference-based checking we need to handle it ourselves. This can be illustrated using a test:

*test/scope_spec.js*

```
it("correctly handles NaNs", function() {
  scope.number = 0/0; // NaN
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.number; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

We're watching a value that happens to be `NaN` and incrementing a counter when it changes. We expect the counter to increment once on the first `$digest` and then stay the same. Instead, as we run the test we're greeted by the "TTL reached" exception. The scope isn't stabilizing because `NaN` is always considered to be different from the last value.

Let's fix that by tweaking the `$$areEqual` function:

*src/scope.js*

```
Scope.prototype.$$areEqual = function(newValue, oldValue, valueEq) {
  if (valueEq) {
    return _.isEqual(newValue, oldValue);
  } else {
    return newValue === oldValue ||
      (typeof newValue === 'number' && typeof oldValue === 'number' &&
       isNaN(newValue) && isNaN(oldValue));
  }
};
```

---

Now the watch behaves as expected with `NaN`s as well.

With the value-checking implementation in place, let's now switch our focus to the ways in which you can interact with a scope from application code.

## $eval - Evaluating Code In The Context of A Scope

There are a few ways in which Angular lets you execute some code in the context of a scope. The simplest of these is `$eval`. It takes a function as an argument and immediately executes that function giving it the scope itself as an argument. It then returns whatever the function returned. `$eval` also takes an optional second argument, which it just passes as-is to the function.

Here are a couple of unit tests showing how one can use `$eval`:

*test/scope_spec.js*

---

```
it("executes $eval'ed function and returns result", function() {
  scope.aValue = 42;

  var result = scope.$eval(function(scope) {
    return scope.aValue;
  });

  expect(result).toBe(42);
});

it("passes the second $eval argument straight through", function() {
  scope.aValue = 42;

  var result = scope.$eval(function(scope, arg) {
    return scope.aValue + arg;
  }, 2);

  expect(result).toBe(44);
});
```

---

  

Implementing `$eval` is straightforward:

*src/scope.js*

```
Scope.prototype.$eval = function(expr, locals) {
  return expr(this, locals);
};
```

So what is the purpose of such a roundabout way to invoke a function? One could argue that `$eval` makes it just slightly more explicit that some piece of code is dealing specifically with the contents of a scope. `$eval` is also a building block for `$apply`, which is what we'll be looking at next.

However, probably the most interesting use of `$eval` only comes when we start discussing *expressions* instead of raw functions. Just like with `$watch`, you can give `$eval` a string expression. It will compile that expression and execute it within the context of the scope. We will implement this in the second part of the book.

## $apply - Integrating External Code With The Digest Cycle

Perhaps the best known of all functions on `Scope` is `$apply`. It is considered the standard way to integrate external libraries to Angular. There's a good reason for this.

`$apply` takes a function as an argument. It executes that function using `$eval`, and then kick-starts the digest cycle by invoking `$digest`. Here's a test case for this:

*test/scope_spec.js*

```
it("executes $apply'ed function and starts the digest", function() {
  scope.aValue = 'someValue';
  scope.counter = 0;

  scope.$watch(
    function(scope) {
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$apply(function(scope) {
    scope.aValue = 'someOtherValue';
  });
  expect(scope.counter).toBe(2);

});
```

  

We have a watch that's watching `scope.aValue` and incrementing a counter. We test that the watch is executed immediately when `$apply` is invoked.

Here is a simple implementation that makes the test pass:

*src/scope.js*

```
Scope.prototype.$apply = function(expr) {
  try {
    return this.$eval(expr);
  } finally {
    this.$digest();
  }
};
```

The `$digest` call is done in a `finally` block to make sure the digest will happen even if the supplied function throws an exception.

The big idea of `$apply` is that we can execute some code that isn't aware of Angular. That code may still change things on the scope, and as long as we wrap the code in `$apply` we can be sure that any watches on the scope will pick up on those changes. When people talk about integrating code to the "Angular lifecycle" using `$apply`, this is essentially what they mean. There really isn't much more to it than that.

## $evalAsync - Deferred Execution

In JavaScript it's very common to execute a piece of code "later" – to defer its execution to some point in the future when the current execution context has finished. The usual way to do this is by calling `setTimeout()` with a zero (or very small) delay parameter.

This pattern applies to Angular applications as well, though the preferred way to do it is by using the `$timeout` service that, among other things, integrates the delayed function to the digest cycle with `$apply`.

But there is also another way to defer code in Angular, and that's the `$evalAsync` function on `Scope`. `$evalAsync` takes a function and schedules it to run later *but* still during the ongoing digest. You can, for example, defer some code from within a watch listener function, knowing that while that code is deferred, it'll still be invoked within the current digest iteration.

The reason why `$evalAsync` is often preferable to a `$timeout` with zero delay has to do with the browser event loop. When you use `$timeout` to schedule some work, you relinquish control to the browser, and let it decide when to run the scheduled work. The browser may then choose to execute other work before it gets to your timeout. It may, for example, render the UI, run click handlers, or process Ajax responses. `$evalAsync`, on the other hand, is much more strict about when the scheduled work is executed. Since it'll happen during the ongoing digest, it's guaranteed to run before the browser decides to do anything else. This difference between `$timeout` and `$evalAsync` is especially significant when you

  

want to prevent unnecessary rendering: Why let the browser render DOM changes that are going to be immediately overridden anyway?

Here's the contract of `$evalAsync` expressed as a unit test:

*test/scope_spec.js*

```
it("executes $evalAsync'ed function later in the same cycle", function() {
  scope.aValue = [1, 2, 3];
  scope.asyncEvaluated = false;
  scope.asyncEvaluatedImmediately = false;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.$evalAsync(function(scope) {
        scope.asyncEvaluated = true;
      });
      scope.asyncEvaluatedImmediately = scope.asyncEvaluated;
    }
  );

  scope.$digest();
  expect(scope.asyncEvaluated).toBe(true);
  expect(scope.asyncEvaluatedImmediately).toBe(false);

});
```

We call `$evalAsync` in the watcher's listener function, and then check that the function was executed during the same digest, but *after* the listener function had finished executing.

The first thing we need is a way to store the `$evalAsync` jobs that have been scheduled. We can do this with an array, which we initialize in the `Scope` constructor:

*src/scope.js*

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
}
```

Let's next define `$evalAsync`, so that it adds the function to execute on this queue:

*src/scope.js*

```
Scope.prototype.$evalAsync = function(expr) {
  this.$$asyncQueue.push({scope: this, expression: expr});
};
```

  

> The reason we explicitly store the current scope in the queued object is related to scope inheritance, which we'll discuss in the next chapter.

We've added bookkeeping for the functions that are to be executed, but we still need to actually execute them. That will happen in `$digest`: The first thing we do in `$digest` is consume everything from this queue and invoke all the deferred functions using `$eval` on the scope that was attached to the async task:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

The implementation guarantees that if you defer a function while the scope is still dirty, the function will be invoked later but still during the same digest. That satisfies our unit test.

## Scheduling $evalAsync from Watch Functions

In the previous section we saw how a function scheduled from a listener function using `$evalAsync` will be executed in the same digest loop. But what happens if you schedule an `$evalAsync` from a *watch function*? Granted, this is something one should not do, since watch function are supposed to be side-effect free. But it is still *possible* to do it, so we should make sure it doesn't wreak havoc on the digest.

If we consider a situation where a watch function schedules an `$evalAsync` *once*, everything seems to be in order. The following test case passes with our current implementation:

*test/scope_spec.js*

```
it("executes $evalAsync'ed functions added by watch functions", function() {
  scope.aValue = [1, 2, 3];
  scope.asyncEvaluated = false;

  scope.$watch(
    function(scope) {
      if (!scope.asyncEvaluated) {
        scope.$evalAsync(function(scope) {
          scope.asyncEvaluated = true;
        });
      }
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  scope.$digest();

  expect(scope.asyncEvaluated).toBe(true);
});
```

---

So what's the problem? As we have seen, we keep running the digest loop as long as there is at least one watch that is dirty. In the test case above, this is the case in the first iteration, when we first return `scope.aValue` from the watch function. That causes the digest to go into the next iteration, during which it also runs the function we scheduled using `$evalAsync`. But what if we schedule an `$evalAsync` when no watch is actually dirty?

*test/scope_spec.js*

---

```
it("executes $evalAsync'ed functions even when not dirty", function() {
  scope.aValue = [1, 2, 3];
  scope.asyncEvaluatedTimes = 0;

  scope.$watch(
    function(scope) {
      if (scope.asyncEvaluatedTimes < 2) {
        scope.$evalAsync(function(scope) {
          scope.asyncEvaluatedTimes++;
        });
      }
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  scope.$digest();

  expect(scope.asyncEvaluatedTimes).toBe(2);
});
```

  

This version does the `$evalAsync` twice. On the second time, the watch function won't be dirty since `scope.aValue` hasn't changed. That means the `$evalAsync` also doesn't run since the `$digest` has terminated. While it would be run on the *next* digest, we really want it to run during this one. That means we need to tweak the termination condition in `$digest` to also see whether there's something in the async queue:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
};
```

The test passes, but now we've introduced another problem. What if a watch function *always* schedules something using `$evalAsync`? We might expect it to cause the iteration limit to be reached, but it does not:

*test/scope_spec.js*

```
it("eventually halts $evalAsyncs added by watches", function() {
  scope.aValue = [1, 2, 3];

  scope.$watch(
    function(scope) {
      scope.$evalAsync(function(scope) { });
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  expect(function() { scope.$digest(); }).toThrow();
});
```

This test case will run forever, since the `while` loop in `$digest` never terminates. What we need to do is also check the status of the async queue in our TTL check:

*src/scope.js*

  

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
};
```

Now we can be sure the digest will terminate, regardless of whether it's running because it's dirty or because there's something in its async queue.

## Scope Phases

Another thing `$evalAsync` does is to schedule a `$digest` if one isn't already ongoing. That means, whenever you call `$evalAsync` you can be sure the function you're deferring will be invoked "very soon" instead of waiting for something else to trigger a digest.

> Though `$evalAsync` does schedule a `$digest`, the preferred way to asynchronously execute code within a digest is using `$applyAsync`, introduced in the next section.

For this to work there needs to be some way for `$evalAsync` to check whether a `$digest` is already ongoing, because in that case it won't bother scheduling one. For this purpose, Angular scopes implement something called a *phase*, which is simply a string attribute in the scope that stores information about what's currently going on.

As a unit test, let's make an expectation about a field called `$$phase`, which should be "$digest" during a digest, "$apply" during an apply function invocation, and `null` otherwise:

*test/scope_spec.js*

```
it("has a $$phase field whose value is the current digest phase", function() {
  scope.aValue = [1, 2, 3];
  scope.phaseInWatchFunction = undefined;
  scope.phaseInListenerFunction = undefined;
  scope.phaseInApplyFunction = undefined;

  scope.$watch(
    function(scope) {
```

  

```
      scope.phaseInWatchFunction = scope.$$phase;
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      scope.phaseInListenerFunction = scope.$$phase;
    }
  );

  scope.$apply(function(scope) {
    scope.phaseInApplyFunction = scope.$$phase;
  });

  expect(scope.phaseInWatchFunction).toBe('$digest');
  expect(scope.phaseInListenerFunction).toBe('$digest');
  expect(scope.phaseInApplyFunction).toBe('$apply');

});
```

---

> We don't need to explicitly call `$digest` here to digest the scope, because invoking `$apply` does it for us.

In the `Scope` constructor, let's introduce the `$$phase` field, setting it initially as `null`:

*src/scope.js*

---

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$phase = null;
}
```

---

Next, let's define a couple of functions for controlling the phase: One for setting it and one for clearing it. Let's also add an additional check to make sure we're not trying to set a phase when one is already active:

*src/scope.js*

---

```
Scope.prototype.$beginPhase = function(phase) {
  if (this.$$phase) {
    throw this.$$phase + ' already in progress.';
  }
  this.$$phase = phase;
};

Scope.prototype.$clearPhase = function() {
  this.$$phase = null;
};
```

  

In `$digest`, let's now set the phase as `"$digest"` for the duration of the outer digest loop:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase("$digest");
  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      this.$clearPhase();
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();
};
```

Let's also tweak `$apply` so that it also sets the phase for itself:

*src/scope.js*

```
Scope.prototype.$apply = function(expr) {
  try {
    this.$beginPhase("$apply");
    return this.$eval(expr);
  } finally {
    this.$clearPhase();
    this.$digest();
  }
};
```

And finally we can add the scheduling of the `$digest` into `$evalAsync`. Let's first define the requirement as a unit test:

*test/scope_spec.js*

  

```
it("schedules a digest in $evalAsync", function(done) {
  scope.aValue = "abc";
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$evalAsync(function(scope) {
  });

  expect(scope.counter).toBe(0);
  setTimeout(function() {
    expect(scope.counter).toBe(1);
    done();
  }, 50);
});
```

We check that the digest is indeed run, not immediately during the `$evalAsync` call but just slightly after it. Our definition of "slightly after" here is after a 50 millisecond timeout. To make `setTimeout` work with Jasmine, we use its asynchronous test support: The test case function takes an optional `done` callback argument, and will only finish once we have called it, which we do after the timeout.

The `$evalAsync` function can now check the current phase of the scope, and if there isn't one (and no async tasks have been scheduled yet), schedule the digest.

*src/scope.js*

```
Scope.prototype.$evalAsync = function(expr) {
  var self = this;
  if (!self.$$phase && !self.$$asyncQueue.length) {
    setTimeout(function() {
      if (self.$$asyncQueue.length) {
        self.$digest();
      }
    }, 0);
  }
  self.$$asyncQueue.push({scope: self, expression: expr});
};
```

With this implementation, when you invoke `$evalAsync` you can be sure a digest will happen in the near future, regardless of when or where you invoke it.

If you call `$evalAsync` when a digest is already running, your function will be evaluated during that digest. If there is no digest running, one is started. We use `setTimeout` to defer the beginning of the digest slightly. This way callers of `$evalAsync` can be ensured the function will always return immediately instead of evaluating the expression synchronously, regardless of the current status of the digest cycle.

  

## Coalescing $apply Invocations - $applyAsync

While `$evalAsync` can be used to defer work either from inside a digest or from outside one, it is really designed for the former use case. The digest-launching `setTimeout` call is there mostly just to prevent confusion if someone was to call `$evalAsync` from outside a digest.

For the use case of `$applying` a function from outside a digest loop asynchronously, there is also a specialized function called `$applyAsync`. It is designed to be used like `$apply` is - for integrating code that's not aware of the Angular digest cycle. Unlike `$apply`, it does not evaluate the given function immediately nor does it launch a digest immediately. Instead, it schedules both of these things to happen after a short period of time.

The original motivation for adding `$applyAsync` was handling HTTP responses: Whenever the `$http` service receives a response, any response handlers are invoked and a digest is launched. That means a digest is run for each HTTP response. This may cause performance problems with applications that have a lot of HTTP traffic (as many apps do during startup) and/or an expensive digest cycle. The `$http` service can now be configured to use `$applyAsync` instead, in which case HTTP responses arriving very close to each other will be coalesced into a single digest. However, `$applyAsync` is in no way tied to the `$http` service, and you can use it for anything that might benefit from coalescing digests.

As we see in our first test for it, when we `$applyAsync` a function, it does not immediately cause anything to happen, but 50 milliseconds later it will have:

*test/scope_spec.js*

```
it('allows async $apply with $applyAsync', function(done) {
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$applyAsync(function(scope) {
    scope.aValue = 'abc';
  });
  expect(scope.counter).toBe(1);

  setTimeout(function() {
    expect(scope.counter).toBe(2);
    done();
  }, 50);
});
```

  

So far this is no different from `$evalAsync`, but we start to see the difference when we call `$applyAsync` from a listener function. If we used `$evalAsync`, the function would still be called during the same digest. But `$applyAsync` *always* defers the invocation:

*test/scope_spec.js*

```
it("never executes $applyAsync'ed function in the same cycle", function(done) {
  scope.aValue = [1, 2, 3];
  scope.asyncApplied = false;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.$applyAsync(function(scope) {
        scope.asyncApplied = true;
      });
    }
  );

  scope.$digest();
  expect(scope.asyncApplied).toBe(false);
  setTimeout(function() {
    expect(scope.asyncApplied).toBe(true);
    done();
  }, 50);
});
```

Let's begin the implementation of `$applyAsync` by introducing another queue in the Scope constructor. This is for work that has been scheduled with `$applyAsync`:

*src/scope.js*

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$phase = null;
}
```

When someone calls `$applyAsync`, we'll push a function to the queue. The function will later evaluate the given expression in the context of the scope, just like `$apply` does:

*src/scope.js*

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
});
```

  

What we should also do here is actually schedule the function application. We can do this with `setTimeout` and a delay of 0. In that timeout, we `$apply` a function which drains the queue and invokes all the functions in it:

*src/scope.js*

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  setTimeout(function() {
    self.$apply(function() {
      while (self.$$applyAsyncQueue.length) {
        self.$$applyAsyncQueue.shift()();
      }
    });
  }, 0);
};
```

> Note that we do not `$apply` each individual item in the queue. We only `$apply` once, outside the loop. We only want to digest once here.

As we discussed, the main point of `$applyAsync` is to optimize things that happen in quick succession so that they only need a single digest. We haven't got this exactly right yet. Each call to `$applyAsync` currently schedules a new digest, which is plain to see if we increment a counter in a watch function:

*test/scope_spec.js*

```
it('coalesces many calls to $applyAsync', function(done) {
  scope.counter = 0;

  scope.$watch(
    function(scope) {
      scope.counter++;
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  scope.$applyAsync(function(scope) {
    scope.aValue = 'abc';
  });
  scope.$applyAsync(function(scope) {
    scope.aValue = 'def';
```

  

```
  });

  setTimeout(function() {
    expect(scope.counter).toBe(2);
    done();
  }, 50);
});
```

---

We want the counter to be 2 (the watch is executed twice on the first digest), not any more than that.

What we need to do is keep track of whether a `setTimeout` to drain the queue has already been scheduled. We'll keep this information in a private scope attribute called `$$applyAsyncId`:

*src/scope.js*

---

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$applyAsyncId = null;
  this.$$phase = null;
}
```

---

We can then check this attribute when scheduling the job, and maintain its state when the job is scheduled and when it finishes:

*src/scope.js*

---

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  if (self.$$applyAsyncId === null) {
    self.$$applyAsyncId = setTimeout(function() {
      self.$apply(function() {
        while (self.$$applyAsyncQueue.length) {
          self.$$applyAsyncQueue.shift()();
        }
        self.$$applyAsyncId = null;
      });
    }, 0);
  }
};
```

---

  

Another aspect of `$applyAsync` is that it should not do a digest if one happens to be launched for some other reason before the timeout triggers. In those cases the digest should drain the queue and the `$applyAsync` timeout should be cancelled:

*test/scope_spec.js*

```
it('cancels and flushes $applyAsync if digested first', function(done) {
  scope.counter = 0;

  scope.$watch(
    function(scope) {
      scope.counter++;
      return scope.aValue;
    },
    function(newValue, oldValue, scope) { }
  );

  scope.$applyAsync(function(scope) {
    scope.aValue = 'abc';
  });
  scope.$applyAsync(function(scope) {
    scope.aValue = 'def';
  });

  scope.$digest();
  expect(scope.counter).toBe(2);
  expect(scope.aValue).toEqual('def');

  setTimeout(function() {
    expect(scope.counter).toBe(2);
    done();
  }, 50);
});
```

Here we test that everything we have scheduled with `$applyAsync` happens immediately if we call `$digest`. That leaves nothing to be done later.

Let's first extract the flushing of the queue out of the inner function in `$applyAsync` itself, so that we can call it from multiple locations:

*src/scope.js*

```
Scope.prototype.$$flushApplyAsync = function() {
  while (this.$$applyAsyncQueue.length) {
    this.$$applyAsyncQueue.shift()();
  }
  this.$$applyAsyncId = null;
};
```

*src/scope.js*

  

```
Scope.prototype.$applyAsync = function(expr) {
  var self = this;
  self.$$applyAsyncQueue.push(function() {
    self.$eval(expr);
  });
  if (self.$$applyAsyncId === null) {
    self.$$applyAsyncId = setTimeout(function() {
      self.$apply(_.bind(self.$$flushApplyAsync, self));
    }, 0);
  }
};
```

> The LoDash `_.bind` function is equivalent to ECMAScript 5 Function.prototype.bind, and is used to make sure the `this` receiver of the function is a known value.

Now we can also call this function from `$digest` - if there's an `$applyAsync` flush timeout currently pending, we cancel it and flush the work immediately:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase("$digest");

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();
};
```

And that's it for `$applyAsync`. It is a useful little optimization for situations where you need to `$apply`, but know you'll be doing it several times within a short period of time.

  

# Running Code After A Digest - $$postDigest

There's one more way you can attach some code to run in relation to the digest cycle, and that's by scheduling a `$$postDigest` function.

The double dollar sign in the name of the function hints that this is really an internal facility for Angular, rather than something application developers should use. But it is there, so we'll also implement it.

Just like `$evalAsync` and `$applyAsync`, `$$postDigest` schedules a function to run "later". Specifically, the function will be run *after* the next digest has finished. Similarly to `$evalAsync`, a function scheduled with `$$postDigest` is executed just once. Unlike `$evalAsync` or `$applyAsync`, scheduling a `$$postDigest` function does *not* cause a digest to be scheduled, so the function execution is delayed until the digest happens for some other reason. Here's a unit test the specifies these requirements:

*test/scope_spec.js*

```
it("runs a $$postDigest function after each digest", function() {
  scope.counter = 0;

  scope.$$postDigest(function() {
    scope.counter++;
  });

  expect(scope.counter).toBe(0);

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.$digest();
  expect(scope.counter).toBe(1);

});
```

As the name implies, `$$postDigest` functions run *after* the digest, so if you make changes to the scope from within `$$postDigest` they won't be immediately picked up by the dirty-checking mechanism. If that's what you want, you should call `$digest` or `$apply` manually:

*test/scope_spec.js*

```
it("does not include $$postDigest in the digest", function() {
  scope.aValue = 'original value';

  scope.$$postDigest(function() {
    scope.aValue = 'changed value';
  });
  scope.$watch(
    function(scope) {
```

  

```
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      scope.watchedValue = newValue;
    }
  );

  scope.$digest();
  expect(scope.watchedValue).toBe('original value');

  scope.$digest();
  expect(scope.watchedValue).toBe('changed value');

});
```

---

To implement `$$postDigest` let's first initialize one more array in the `Scope` constructor:

*src/scope.js*

---

```
function Scope() {
  this.$$watchers = [];
  this.$$lastDirtyWatch = null;
  this.$$asyncQueue = [];
  this.$$applyAsyncQueue = [];
  this.$$applyAsyncId = null;
  this.$$postDigestQueue = [];
  this.$$phase = null;
}
```

---

Next, let's implement `$$postDigest` itself. All it does is add the given function to the queue:

*src/scope.js*

---

```
Scope.prototype.$$postDigest = function(fn) {
  this.$$postDigestQueue.push(fn);
};
```

---

Finally, in `$digest`, let's drain the queue and invoke all those functions once the digest has finished:

*src/scope.js*

---

  

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase("$digest");

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  do {
    while (this.$$asyncQueue.length) {
      var asyncTask = this.$$asyncQueue.shift();
      asyncTask.scope.$eval(asyncTask.expression);
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      this.$clearPhase();
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();

  while (this.$$postDigestQueue.length) {
    this.$$postDigestQueue.shift()();
  }
};
```

We consume the queue by removing functions from the beginning of the array using `Array.shift()` until the array is empty, and by immediately executing those functions. `$$postDigest` functions are not given any arguments.

## Handling Exceptions

Our `Scope` implementation is becoming something that resembles the one in Angular. It is, however, quite brittle. That's mainly because we haven't put much thought into exception handling.

If an exception occurs in a watch function, an `$evalAsync` or `$applyAsync` function, or a `$$postDigest` function, our current implementation will just give up and stop whatever it's doing. Angular's implementation, however, is actually much more robust than that. Exceptions thrown before, during, or after a digest are caught and logged, and the operation then resumed where it left off.

> Angular actually forwards exceptions to a special `$exceptionHandler` service. Since we don't have such a service yet, we'll simply log the exceptions to the console for now.

  

In watches there are two points when exceptions can happen: In the watch functions and in the listener functions. In either case, we want to log the exception and continue with the next watch as if nothing had happened. Here are two test cases for the two functions:

*test/scope_spec.js*

```
it("catches exceptions in watch functions and continues", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { throw "error"; },
    function(newValue, oldValue, scope) { }
  );
  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});

it("catches exceptions in listener functions and continues", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      throw "Error";
    }
  );
  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

In both cases we define two watches, the first of which throws an exception. We check that the second watch is still executed.

To make these tests pass we need to modify the `$$digestOnce` function and wrap the execution of each watch in a `try...catch` clause:

*src/scope.js*

  

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEach(this.$$watchers, function(watcher) {
    try {
      newValue = watcher.watchFn(self);
      oldValue = watcher.last;
      if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
        self.$$lastDirtyWatch = watcher;
        watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
        watcher.listenerFn(newValue,
          (oldValue === initWatchVal ? newValue : oldValue),
          self);
        dirty = true;
      } else if (self.$$lastDirtyWatch === watcher) {
        return false;
      }
    } catch (e) {
      console.error(e);
    }
  });
  return dirty;
};
```

What remains is exception handling in `$evalAsync`, `$applyAsync`, and `$$postDigest`. All of them are used to execute arbitrary functions in relation to the digest loop. In none of them do we want an exception to cause the loop to end prematurely.

For `$evalAsync` we can define a test case that checks that a watch is run even when an exception is thrown from one of the functions scheduled for `$evalAsync`:

*test/scope_spec.js*

```
it("catches exceptions in $evalAsync", function(done) {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$evalAsync(function(scope) {
    throw "Error";
  });
```

  

```
  setTimeout(function() {
    expect(scope.counter).toBe(1);
    done();
  }, 50);

});
```

---

For `$applyAsync`, we'll define a test case checking that a function scheduled with `$applyAsync` is invoked even if it has functions before it that throw exceptions.

*test/scope_spec.js*

---

```
it("catches exceptions in $applyAsync", function(done) {
  scope.$applyAsync(function(scope) {
    throw "Error";
  });
  scope.$applyAsync(function(scope) {
    throw "Error";
  });
  scope.$applyAsync(function(scope) {
    scope.applied = true;
  });

  setTimeout(function() {
    expect(scope.applied).toBe(true);
    done();
  }, 50);
});
```

---

> We use two error-throwing functions, because if we used just one, the second function would indeed run. That's because `$apply` launches `$digest`, and the `$applyAsync` queue drainage therein from a `finally` block.

For `$$postDigest` the digest will already be over, so it doesn't make sense to test it with a watch. We can test it with a second `$$postDigest` function instead, making sure it also executes:

*test/scope_spec.js*

---

```
it("catches exceptions in $$postDigest", function() {
  var didRun = false;

  scope.$$postDigest(function() {
    throw "Error";
  });
  scope.$$postDigest(function() {
    didRun = true;
```

```
  });

  scope.$digest();
  expect(didRun).toBe(true);
});
```

Fixing both `$evalAsync` and `$$postDigest` involves changing the `$digest` function. In both cases we wrap the function execution in `try...catch`:

*src/scope.js*

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  this.$$lastDirtyWatch = null;
  this.$beginPhase('$digest');

  if (this.$$applyAsyncId) {
    clearTimeout(this.$$applyAsyncId);
    this.$$flushApplyAsync();
  }

  do {
    while (this.$$asyncQueue.length) {
      try {
        var asyncTask = this.$$asyncQueue.shift();
        asyncTask.scope.$eval(asyncTask.expression);
      } catch (e) {
        console.error(e);
      }
    }
    dirty = this.$$digestOnce();
    if ((dirty || this.$$asyncQueue.length) && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty || this.$$asyncQueue.length);
  this.$clearPhase();

  while (this.$$postDigestQueue.length) {
    try {
      this.$$postDigestQueue.shift()();
    } catch (e) {
      console.error(e);
    }
  }
};
```

Fixing `$applyAsync`, on the other hand, is done in the loop that drains the queue in `$$flushApplyAsync`:

*src/scope.js*

  

```
Scope.prototype.$$flushApplyAsync = function() {
  while (this.$$applyAsyncQueue.length) {
    try {
      this.$$applyAsyncQueue.shift()();
    } catch (e) {
      console.error(e);
    }
  }
  this.$$applyAsyncId = null;
};
```

Our digest cycle is now a lot more robust when it comes to exceptions.

## Destroying A Watch

When you register a watch, most often you want it to stay active as long as the scope itself does, so you don't ever really explicitly remove it. There are cases, however, where you want to destroy a particular watch while still keeping the scope operational. That means we need a removal operation for watches.

The way Angular implements this is actually quite clever: The `$watch` function in Angular has a return value. It is a function that, when invoked, destroys the watch that was registered. If a user wants to be able to remove a watch later, they just need to keep hold of the function returned when they registered the watch, and then call it once the watch is no longer needed:

*test/scope_spec.js*

```
it("allows destroying a $watch with a removal function", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  var destroyWatch = scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);

  scope.aValue = 'def';
  scope.$digest();
  expect(scope.counter).toBe(2);

  scope.aValue = 'ghi';
```

```
  destroyWatch();
  scope.$digest();
  expect(scope.counter).toBe(2);
});
```

---

To implement this, we need to return a function that removes the watch from the `$$watchers` array:

*src/scope.js*

---

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    valueEq: !!valueEq,
    last: initWatchVal
  };
  self.$$watchers.push(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
    }
  };
};
```

---

While that takes care of the watch removal itself, there are a few corner cases we need to deal with before we have a robust implementation. They all have to do with the not too uncommon use case of removing a watch *during a digest*.

First of all, a watch might remove *itself* in its own watch or listener function. This should not affect other watches:

*test/scope_spec.js*

---

```
it("allows destroying a $watch during digest", function() {
  scope.aValue = 'abc';

  var watchCalls = [];

  scope.$watch(
    function(scope) {
      watchCalls.push('first');
      return scope.aValue;
    }
  );
```

```
  var destroyWatch = scope.$watch(
    function(scope) {
      watchCalls.push('second');
      destroyWatch();
    }
  );

  scope.$watch(
    function(scope) {
      watchCalls.push('third');
      return scope.aValue;
    }
  );

  scope.$digest();
  expect(watchCalls).toEqual(['first', 'second', 'third', 'first', 'third']);
});
```

---

In the test we have three watches. The middlemost watch removes itself when it is first called, leaving only the first and the third watch. We verify that the watches are iterated in the correct order: During the first turn of the loop each watch is executed once. Then, since the digest was dirty, each watch is executed again, but this time the second watch is no longer there.

What's happening instead is that when the second watch removes itself, the watch collection gets shifted to the left, causing `$$digestOnce` to skip the third watch during that round.

The trick is to reverse the `$$watches` array, so that new watches are added to the beginning of it and iteration is done from the end to the beginning. When a watcher is then removed, the part of the watch array that gets shifted has already been handled during that digest iteration and it won't affect the rest of it.

When adding a watch, we should use `Array.unshift` instead of `Array.push`:

*src/scope.js*

---

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn || function() { },
    last: initWatchVal,
    valueEq: !!valueEq
  };
  this.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
```

```
    }
  };
};
```

---

Then, when iterating, we should use `_.forEachRight` instead of `_.forEach` to reverse the iteration order:

*src/scope.js*

---

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEachRight(this.$$watchers, function(watcher) {
    try {
      newValue = watcher.watchFn(self);
      oldValue = watcher.last;
      if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
        self.$$lastDirtyWatch = watcher;
        watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
        watcher.listenerFn(newValue,
          (oldValue === initWatchVal ? newValue : oldValue),
          self);
        dirty = true;
      } else if (self.$$lastDirtyWatch === watcher) {
        return false;
      }
    } catch (e) {
      console.error(e);
    }
  });
  return dirty;
};
```

---

The next case is a watch removing *another watch*. Observe the following test case:

*src/scope.js*

---

```
it("allows a $watch to destroy another during digest", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  scope.$watch(
    function(scope) {
      return scope.aValue;
    },
    function(newValue, oldValue, scope) {
      destroyWatch();
    }
```

  

```
  );

  var destroyWatch = scope.$watch(
    function(scope) { },
    function(newValue, oldValue, scope) { }
  );

  scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(1);
});
```

---

This test case fails. The culprit is our short-circuiting optimization. Recall that in `$$digestOnce` we see whether the current watch was the last dirty one seen and is now clean. If so, we end the digest. What happens in this test case is:

1. The first watch is executed. It is dirty, so it is stored in `$$lastDirtyWatch` and its listener is executed. The listener destroys the second watch.
2. The first watch is executed *again*, because it has moved one position down in the watcher array. This time it is clean, and since it is also in `$$lastDirtyWatch`, the digest ends. We never get to the third watch.

We should eliminate the short-circuiting optimization on watch removal so that this does not happen:

*src/scope.js*

---

```
Scope.prototype.$watch = function(watchFn, listenerFn, valueEq) {
  var self = this;
  var watcher = {
    watchFn: watchFn,
    listenerFn: listenerFn,
    valueEq: !!valueEq,
    last: initWatchVal
  };
  self.$$watchers.unshift(watcher);
  this.$$lastDirtyWatch = null;
  return function() {
    var index = self.$$watchers.indexOf(watcher);
    if (index >= 0) {
      self.$$watchers.splice(index, 1);
      self.$$lastDirtyWatch = null;
    }
  };
};
```

  

The final case to consider is when a watch removes *several watches* when executed:

*test/scope_spec.js*

```
it("allows destroying several $watches during digest", function() {
  scope.aValue = 'abc';
  scope.counter = 0;

  var destroyWatch1 = scope.$watch(
    function(scope) {
      destroyWatch1();
      destroyWatch2();
    }
  );

  var destroyWatch2 = scope.$watch(
    function(scope) { return scope.aValue; },
    function(newValue, oldValue, scope) {
      scope.counter++;
    }
  );

  scope.$digest();
  expect(scope.counter).toBe(0);
});
```

The first watch destroys not only itself, but also a second watch that would have been executed next. While we don't expect that second watch to execute, we don't expect an exception to be thrown either, which is what actually happens.

What we need to do is check that the current watch actually exists while we're iterating:

*src/scope.js*

```
Scope.prototype.$$digestOnce = function() {
  var self = this;
  var newValue, oldValue, dirty;
  _.forEachRight(this.$$watchers, function(watcher) {
    try {
      if (watcher) {
        newValue = watcher.watchFn(self);
        oldValue = watcher.last;
        if (!self.$$areEqual(newValue, oldValue, watcher.valueEq)) {
          self.$$lastDirtyWatch = watcher;
          watcher.last = (watcher.valueEq ? _.cloneDeep(newValue) : newValue);
          watcher.listenerFn(newValue,
            (oldValue === initWatchVal ? newValue : oldValue),
            self);
```

  

```
          dirty = true;
        } else if (self.$$lastDirtyWatch === watcher) {
          return false;
        }
      }
    } catch (e) {
      console.error(e);
    }
  });
  return dirty;
};
```

---

And finally we can rest assured our digest will keep on running regardless of watches being removed.

## Watching Several Changes With One Listener: `$watchGroup`

So far we have been looking at watches and listeners as simple cause-and-effect pairs: When this changes, do that. It is not unusual, however, to want to watch *several* pieces of state and execute some code when *any one* of them changes.

Since Angular watches are just normal JavaScript functions, this is perfectly doable with the watch implementation we already have: Just craft a watch function that runs multiple checks and returns some combined value of them that causes the listener to fire.

As it happens, from Angular 1.3 onwards it is not necessary to craft these kinds of functions manually. Instead, you can use a built-in Scope feature called `$watchGroup`.

The `$watchGroup` function takes several watch functions wrapped in an array, and a single listener function. The idea is that when any of the watch functions given in the array detects a change, the listener function is invoked. The listener function is given the new and old values of the watches wrapped in arrays, in the order of the original watch functions.

Here's the first test case for this, wrapped in a new `describe` block:

*test/scope_spec.js*

---

```
describe('$watchGroup', function() {

  var scope;
  beforeEach(function() {
    scope = new Scope();
  });

  it('takes watches as an array and calls listener with arrays', function() {
    var gotNewValues, gotOldValues;

    scope.aValue = 1;
```

```
    scope.anotherValue = 2;

    scope.$watchGroup([
      function(scope) { return scope.aValue; },
      function(scope) { return scope.anotherValue; }
    ], function(newValues, oldValues, scope) {
      gotNewValues = newValues;
      gotOldValues = oldValues;
    });
    scope.$digest();

    expect(gotNewValues).toEqual([1, 2]);
    expect(gotOldValues).toEqual([1, 2]);
  });

});
```

---

In the test we grab the `newValues` and `oldValues` arguments received by the listener, and check that they are arrays containing the return values of the watch functions.

Let's take a first stab at implementing `$watchGroup`. We could try to just register each watch individually, reusing the listener for each one:

*src/scope.js*

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
    var self = this;
    _.forEach(watchFns, function(watchFn) {
        self.$watch(watchFn, listenerFn);
    });
};
```

---

This doesn't quite cut it though. We expect the listener to receive *arrays* of all the watch values, but now it just gets called with each watch value individually.

We'll need to define a separate internal listener function for each watch, and inside those internal listeners collect the values into arrays. We can then give those arrays to the original listener function. We'll use one array for the new values and another for the old values:

*src/scope.js*

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var newValues = new Array(watchFns.length);
  var oldValues = new Array(watchFns.length);
  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      listenerFn(newValues, oldValues, self);
    });
  });
};
```

  

`$watchGroup` always uses reference watches for change detection.

The problem with our first implementation is that it calls the listener a bit too eagerly: If there are several changes in the watch array, the listener will get called several times, and we'd like for it to get called just once. Even worse, since we're calling the listener immediately upon noticing a change, it's likely that we have a mixture of new and previous values in our `oldValues` and `newValues` arrays, causing the user to see an inconsistent combination of values.

Let's test that the listener is called just once even in the presence of multiple changes:

*test/scope_spec.js*

```
it('only calls listener once per digest', function() {
  var counter = 0;

  scope.aValue = 1;
  scope.anotherValue = 2;

  scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    counter++;
  });
  scope.$digest();

  expect(counter).toEqual(1);
});
```

How can we defer the listener call to a moment when all watches will have been checked? Since in `$watchGroup` we're not in charge of running the digest, there's no obvious place for us to put the listener call. But what we can do is use the `$evalAsync` function we implemented earlier in the chapter. Its purpose is to do some work later but still during the same digest - just the ticket for us!

We'll create a new internal function in `$watchGroup`, called `watchGroupListener`. This is the function that's in charge of calling the original listener with the two arrays. Then, in each individual listener we schedule a call to this function *unless one has been scheduled already*:

*src/scope.js*

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
```

  

```
  var changeReactionScheduled = false;

  function watchGroupListener() {
    listenerFn(newValues, oldValues, self);
    changeReactionScheduled = false;
  }

  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
    });
  });
};
```

---

That takes care of the basic behavior of `$watchGroup`, and we can turn our attention to a couple of special cases.

One issue is related to the requirement that when a listener is called for the very first time, both the new and old values should be the same. Now, our `$watchGroup` already does something like this, because it's built on the `$watch` function that implements this behavior. On the first invocation, the contents of the `newValues` and `oldValues` arrays will be exactly the same.

However, while the *contents* of those two arrays are the same, they are currently still two separate *array objects*. That breaks the contract of using the same exact value twice. It also means that if a user wants to compare the two values, they cannot use reference equality (`===`), but instead have to iterate the array contents and see if they match.

We want to do better, and have both the old and new values be the *same exact value* on the first invocation:

*test/scope_spec.js*

---

```
it('uses the same array of old and new values on first run', function() {
  var gotNewValues, gotOldValues;

  scope.aValue = 1;
  scope.anotherValue = 2;

  scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    gotNewValues = newValues;
    gotOldValues = oldValues;
  });
```

  

```
  scope.$digest();

  expect(gotNewValues).toBe(gotOldValues);
});
```

While doing this, let's also make sure we won't break what we already have, by adding a test that ensures we still get *different* arrays on subsequent listener invocations:

*test/scope_spec.js*

```
it('uses different arrays for old and new values on subsequent runs', function() {
  var gotNewValues, gotOldValues;

  scope.aValue = 1;
  scope.anotherValue = 2;

  scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    gotNewValues = newValues;
    gotOldValues = oldValues;
  });
  scope.$digest();

  scope.anotherValue = 3;
  scope.$digest();

  expect(gotNewValues).toEqual([1, 3]);
  expect(gotOldValues).toEqual([1, 2]);
});
```

We can implement this requirement by checking in the watch group listener whether it's being called for the first time. If it is, we'll just pass the `newValues` array to the original listener twice:

*src/scope.js*

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
  var changeReactionScheduled = false;
  var firstRun = true;

  function watchGroupListener() {
    if (firstRun) {
      firstRun = false;
```

```
      listenerFn(newValues, newValues, self);
    } else {
      listenerFn(newValues, oldValues, self);
    }
    changeReactionScheduled = false;
  }

  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
    });
  });
};
```

---

The other special case is a situation where the array of watches happens to be empty. It is not completely obvious what to do in this situation. Our current implementation does nothing - if there are no watches, no listeners will get fired. What Angular actually does though is make sure the listener gets called *exactly once*, with empty arrays as the values:

*test/scope_spec.js*

```
it('calls the listener once when the watch array is empty', function() {
  var gotNewValues, gotOldValues;

  scope.$watchGroup([], function(newValues, oldValues, scope) {
    gotNewValues = newValues;
    gotOldValues = oldValues;
  });
  scope.$digest();

  expect(gotNewValues).toEqual([]);
  expect(gotOldValues).toEqual([]);
});
```

---

What we'll do is check for the empty case in `$watchGroup`, schedule a call to the listener, and then return without bothering to do any further setup:

*src/scope.js*

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
```

  

```
  var changeReactionScheduled = false;
  var firstRun = true;

  if (watchFns.length === 0) {
    self.$evalAsync(function() {
      listenerFn(newValues, newValues, self);
    });
    return;
  }

  function watchGroupListener() {
    if (firstRun) {
      firstRun = false;
      listenerFn(newValues, newValues, self);
    } else {
      listenerFn(newValues, oldValues, self);
    }
    changeReactionScheduled = false;
  }

  _.forEach(watchFns, function(watchFn, i) {
    self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
    });
  });
};
```

---

The final feature we'll need for `$watchGroups` is deregistration. One should be able to deregister a watch group in exactly the same way as they deregister an individual watch: By using a removal function returned by `$watchGroup`.

*test/scope_spec.js*

---

```
it('can be deregistered', function() {
  var counter = 0;

  scope.aValue = 1;
  scope.anotherValue = 2;

  var destroyGroup = scope.$watchGroup([
    function(scope) { return scope.aValue; },
    function(scope) { return scope.anotherValue; }
  ], function(newValues, oldValues, scope) {
    counter++;
```

  

```
  });
  scope.$digest();

  scope.anotherValue = 3;
  destroyGroup();
  scope.$digest();

  expect(counter).toEqual(1);
});
```

---

Here we test that once the deregistration function has been called, further changes do not cause the listener to fire.

Since the individual watch registrations already return removal functions, all we really need to do is collect them, and then create a deregistration function that invokes all of them:

*src/scope.js*

---

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
  var self = this;
  var oldValues = new Array(watchFns.length);
  var newValues = new Array(watchFns.length);
  var changeReactionScheduled = false;
  var firstRun = true;

  if (watchFns.length === 0) {
    self.$evalAsync(function() {
      listenerFn(newValues, newValues, self);
    });
    return;
  }

  function watchGroupListener() {
    if (firstRun) {
      firstRun = false;
      listenerFn(newValues, newValues, self);
    } else {
      listenerFn(newValues, oldValues, self);
    }
    changeReactionScheduled = false;
  }

  var destroyFunctions = _.map(watchFns, function(watchFn, i) {
    return self.$watch(watchFn, function(newValue, oldValue) {
      newValues[i] = newValue;
      oldValues[i] = oldValue;
      if (!changeReactionScheduled) {
        changeReactionScheduled = true;
        self.$evalAsync(watchGroupListener);
      }
```

  

```
    });
  });

  return function() {
    _.forEach(destroyFunctions, function(destroyFunction) {
      destroyFunction();
    });
  };
};
```

---

Since we have a special case for the situation where the watch array is empty, it needs its own watch deregistration function as well. The listener is only called once anyway in that situation, but one could still invoke the deregistration function before even the first digest occurs, in which case even that single call should be skipped:

*test/scope_spec.js*

---

```
it('does not call the zero-watch listener when deregistered first', function() {
  var counter = 0;

  var destroyGroup = scope.$watchGroup([], function(newValues, oldValues, scope) {
    counter++;
  });
  destroyGroup();
  scope.$digest();

  expect(counter).toEqual(0);
});
```

---

The deregistration function for this case just sets a boolean flag, which is checked before invoking the listener:

*src/scope.js*

---

```
Scope.prototype.$watchGroup = function(watchFns, listenerFn) {
    var self = this;
    var oldValues = new Array(watchFns.length);
    var newValues = new Array(watchFns.length);
    var changeReactionScheduled = false;
    var firstRun = true;

    if (watchFns.length === 0) {
        var shouldCall = true;
        self.$evalAsync(function() {
            if (shouldCall) {
                listenerFn(newValues, newValues, self);
            }
        });
```

  

```
        return function() {
            shouldCall = false;
        };
    }

    function watchGroupListener() {
        if (firstRun) {
            firstRun = false;
            listenerFn(newValues, newValues, self);
        } else {
            listenerFn(newValues, oldValues, self);
        }
        changeReactionScheduled = false;
    }

    var destroyFunctions = _.map(watchFns, function(watchFn, i) {
        return self.$watch(watchFn, function(newValue, oldValue) {
            newValues[i] = newValue;
            oldValues[i] = oldValue;
            if (!changeReactionScheduled) {
                changeReactionScheduled = true;
                self.$evalAsync(watchGroupListener);
            }
        });
    });

    return function() {
        _.forEach(destroyFunctions, function(destroyFunction) {
            destroyFunction();
        });
    };
};
```

---

## Summary

We've already come a long way, and have a perfectly usable implementation of an Angular-style dirty-checking scope system. In the process you have learned about:

- The two-sided process underlying Angular's dirty-checking: `$watch` and `$digest`.
- The dirty-checking loop and the TTL mechanism for short-circuiting it.
- The difference between reference-based and value-based comparison.
- Executing functions on the digest loop in different ways: Immediately with `$eval` and `$apply` and time-shifted with `$evalAsync`, `$applyAsync`, and `$$postDigest`.
- Exception handling in the Angular digest.
- Destroying watches so they won't get executed again.
- Watching several things with a single effect using the `$watchGroup` function.

  

There is, of course, a lot more to Angular scopes than this. In the next chapter we'll start looking at how scopes can *inherit from other scopes*, and how watches can watch things not only on the scope they are attached to, but also on that scope's parents.

  