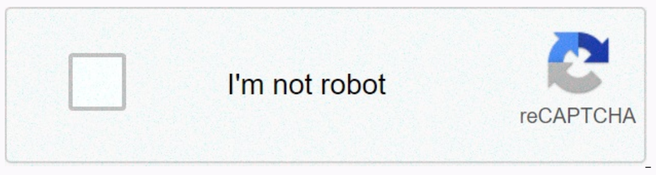

Continue

Angularjs scope variable in template

<! DOCTYPE html><html ng-app=myApp><head><script data-semver=1.5.0 src= amp;gt;</script> <script data-semver=1.5.0 src= amp;gt;</script> <link href=style.css rel=stylesheet> <script src=script.js></script></head><body><h1>AngularJS - $watch to track variable $scope</h1><div> We can mount callbacks to start whenever a specific $scope changes. It gives us access to new and old value. This preview triggers a warning () whenever the number of donations make input changes ($scope.giftTotal). </div><div ng-controller=PersonCtrl><h2>Dospelí</h2><table><tr><th>Meno Narodeniny</th> <th>Happy?</th> <th></th></tr><tr ng-repeat=adult in adults ng-class={'happy': adult.happy, 'sad': !adult.happy}><td>{{adult.name}}</td><td>{{adult.birthday}}</td><td><input type=checkbox ng-model=adult.happy></td></tr></table><p> Zadajte počet darčekov dať: <input type=number ng-model=giftTotal></p><p>Zobraziť počet darčekov dať: </p><pre>{{ dospelí | json }}</pre></div></body></html>

Okamžite aplikáciu , the 'myApp' parameter must correspond to what is in the ng-app var myApp = angular.module ('myApp', ['ngResource']); * Adult is a service that calls rest api * It is not really a REST API, but just call our local .json file as an example * If you call Adult.query(), it will receive adults.json * If you call Adult.get ({}, {aid: 1}), it will receive adult/1.json*/ myApp.factory('Adult', function($resource) { return $resource('adults/:adultId.json', {adultId: '@aid'}); Create a driver, the PersonCtrl parameter must // comply with the ng-controller myApp.controller('PersonCtrl', function ($scope, Adult) { $scope.giftTotal = 0, $scope.$watch('giftTotal', function(newVal, oldVal) { if(newVal === oldVal) { // happens on initial $watch registration return; } console.log('New Val:' + newVal + ' Old Val:' + oldVal); }); // Adults are from the REST API (really static json file) $scope.adults = Adult.query(); body { font size: 24px; } .happy { background color: green; } .sad { background color: red; } #mustache-teen-template-result { display: none; } [{ help: 1, name: adult one, birthday: 1/1/1991, happy: true }, { aid: 2, name: Adult Two, birthday: 2/2/1991, happy: true }, { aid: 3, name: Adult Three, birthday: 3/3/1991, happy: false } { aid: 1, name: Adult One, birthday: 1/1/1991, happy: true } One paging application requires end developers to become better software engineers. CSS and HTML are no longer the biggest problem, in fact there is no longer a single interest. Front-end developer must handle XHRs, application (models, opinions, controllers), performance, animation, styles, structure, SEO, and integration with external services. The result, which results from all of combined is the user experience (UX), which should always be preferred. AngularJS is a very strong framework. It's the third most played repository on GitHub. It is not difficult to start using, but a goal that is designed to achieve demand with understanding. AngularJS developers can no longer ignore memory consumption because navigation will no longer be refreshed. This is the vanguard of web development. Let's hug it! Common error #1: Accessing a range through DOM There are several optimization tweaks recommended for production. One is disabling debugging information. DebugInfoEnabled is a setting that is valid by default and allows access to a range through DOM nodes. To test that through the JavaScript console, select the DOM element and access its range with: ular.element(document.body).scope() This may also be useful when not using jQuery with its CSS, but should not be used outside the console. This is because when the $compileProvider.debugInfoEnabled is set to false, the .scope() call on the DOM node returns undefined. This is one of the few recommended production options. Note that you still have access to the scope through the console, even if you are manufacturing. Call angular.reloadWithDebugInfo() from the console and the application will do just that. Common #2: Don't dot it there You probably read that if you didn't have a dot in your ng-model, you were doing it wrong. When it comes to inheritance, this claim is often true. The ranges have a prototype heritage model, typical of JavaScript, and nested ranges are common to AngularJS. Many directives create child scopes, such as ngRepeat, ngIf, and ngController. When you solve a model, the search runs on the current range and passes through each parent range up to $rootScope. But when setting a new value, what happens depends on what model (variable) we want to change. If the model is primitive, the child range creates only the new model. However, if the change changes to a model object property, a search in the parent ranges finds the referenced object and changes its actual property. The new model should not be set to the current range, so there would be no camouflage: MainController($scope) function { $scope.foo = 1; $scope.bar = {innerProperty: 2}; } angular.module('myApp', []) .controller('MainController', MainController); <div ng-controller=MainController><p>VONKAJŠÍ ROZSAH PÔSOBNOSTI:</p><p>{{ foo }}</p><p>{{ bar.innerProperty }}</p><div ng-if=foo><!— ng-if creates a new scope —><p>VNÚTORNÝ ROZSAH PÔSOBNOSTI</p><p>{{ foo }}</p><p>{{ bar.innerProperty }}</p><button ng-click=foo = 2>Nastavenie primitívnej</button> <button ng-click=bar.innerProperty = 3>muty objektu</button></div></div> Kliknutím na tlačidlo označené Nastaviť primitívne nastaví foo vo vnútornom rozsahu na 2, ale nezmení foo vo vonkajšom rozsahu. Click the Change Object label button changes the bar property from the parent range. Since there is no variable variable internal range, no shielding will happen, and the visible value for the bar will be 3 in both ranges. Another way to do this is to take advantage of the fact that the parent ranges and root range are listed from each range. The $parent and $root objects can be used to access the parent $rootScope and directly from the view. This may be a powerful way, but I'm not a fan of that because of the problem of targeting a particular range into a stream. There is another way to set and access range-specific properties - using the controllerAs syntax. Common error #3: You are not using controller syntax An alternative and most effective way to assign models to use a driver object instead of an $scope. Instead of injecting a range, we can define models like this: MainController($scope) function { this.foo = 1; var that = to; var setBar = function () { // that.bar = {someProperty: 2}; this.bar = {someProperty: 2}; }; setBar.call(toto); // there are other conventions: // var MC = this; // setBar.call(toto); when using 'this' inside setBar() } <div><p>EXTERNAL SCOPE:</p><p>{{ MC.foo }}</p><p >{{ MC.bar.someProperty }}</p><div ng-if=test1><p>INTERNAL SCOPE</p&p gt;<p>{{ MC.foo }}</p><p>{{ MC.bar.someProperty }}</p><button ng-click=MC.foo = 3>Change MC.. foo</button> <button ng-click=MC.bar.someProperty = 5>Change MC.bar.someProperty</button></div></div> This is much less confusing. Especially when there are many nested ranges, as can be the case with nested states. There is more to the controllerAs syntax. Common error #4: There is no full use of the ControllerAs Syntax, there are several warnings with how the controller object is exposed. It's basically an object set to a driver range as well as a normal model. If you need to monitor the property of the driver object, you can monitor the feature, but you are not required. Here's an example: MainController($scope) function { this.title = 'Some title'; $scope.$watch(angular.bind(this, function() { return this.title, }), function (newVal, oldVal) { // handle changes }); } It's easier to do it easily: MainController($scope) { this.title = 'Some name'; $scope.$watch('MC.title', function (newVal, oldVal) { // handle changes }); } This means that even in a range string, you can access the MC from a child controller: NestedController($scope) { if ($scope. MC & $scope. MC.title === 'Some name') { $scope. MC.title = New Name; } } } However, if you want you to do this, you must comply with the shortcut you use for controllers. There are at least three ways to set it up. Have you seen the first: <div ng-controller=MainController as MC> ... </div> However, if you are using a user user system router, entering the controller in this way is prone to error. For states, controllers should be specified in the status configuration: ular.module('myApp', .config(funkcia ($stateProvider) { $stateProvider .state('main', { url: '/', radič: 'MainController as MC', templateUrl: templateUrl: }) }). controller('MainController', function () { ... }); Is there another way to comment: (...) .state('main', { url: '/', controller: 'MainController', controllerAs: 'MC', templateUrl: '/path/to/template.html' }) You can do the same in the guidelines: AnotherController() function { this.text = 'abc'; } Test functionForToptal() { return { controller: 'AnotherController as AC', template: '<p>{{ AC.text }}</p>' }; } angular.module('myApp', []) .controller('AnotherController', AnotherController) .directive('testForToptal', testForToptal); The second way to annotate is also valid, though less concise: test functionForToptal() { return { controller: AnotherController, controllerAs: AC, template: '<p>{{ AC.text }}</p>' }; } Common error #5: We do not use named views with UI-ROUTER to power de facto routing solution for AngularJS was until now, ui-router. Removed from the kernel sometime before, the ngRoute module was too basic for more sophisticated routing. There is a new NgRouter to do so, but the authors still consider it too early to produce. When I write this, the stable angular is 1.3.15, and the ui-router rocks. Main reasons: amazing nesting status route abstraction optional and required parameters Here I will cover the state of nesting to avoid AngularJS errors. Think of it as a complex but standard use case. There is an app that has a home page view and a product view. Product view has three separate sections: introduction, widget, and content. We want the widget to persist and not reload when switching between states. But the content should reload. Zvážte nasledujúcu štruktúru stránky indexu produktov HTML: <body><header> <!-- SOME STATIC HEADER CONTENT --> </header><section class=main><div class=page-content><div class=row><div class=col-xs-12><section class=intro><h2>INTRO NIEKTORÉ ŠPECIFICKÉ PRE VÝROBOK</h2></section></div></div><div class=row><div class=col-xs-3><section class=widget> <!-- some widget, which should never reload --> </section></div><div class=col-xs-9><section class=content><div class=product-content><h2>Názov produktu</h2>Obsah špecifický pre kontext</div></section></div></div></div></section> <footer> <!-- SOME STATIC HEADER CONTENT --> </footer></ body>This is something we could get from the HTML code, and now you need to separate it into files and states. I generally go with the convention that there is an abstract major state that maintains global data when needed. Use it instead of $rootScope. The main state will also maintain the static HTML that is required on each page. I keep the index.html clean. <!— index.html —><body><div ui-view=></div></body><!— main.html —> <header> <!-- SOME STATIC HEADER --> </header><section class=main><div ui-view=></div></section>Potom sa pozrime na stránku indexu produktov: <footer> <!-- SOME STATIC HEADER CONTENT --> </footer><div class=page-content><div class=row><div class=col-xs-12><section class=intro> <div ></div ></section></div></div></div> ></section></div></div></div> <div class=row><div class=col-xs-3><section class=widget><div ui-view=widget></div></section></div><div class=col-xs-9><section class=content><div ui-view=content></div></section></div>>As you can see, the product index page has three named views. One for the intro, one for the widget, and one for the product. We meet the specs! So now let's set the routing: function config($stateProvider) { $stateProvider // MAIN ABSTRACT STATE, ALWAYS ON .state('main, { abstract: true, url: '/', controller: 'MainController as MC', templateUrl: '/routing-demo/main.html' }) // SIMPLE HOMEPAGE .state('main.homepage', { url: '', controller: 'HomepageController as HC', templateUrl: '/routing-demo/homepage.html' }) // ABOVE IS ALL GOOD, HERE IS THE PROBLEM // COMPLEX PRODUCT PAGE .state('main.product', { abstract: true, url: ':id', controller: 'ProductController as PC', templateUrl: '/routing-demo/product.html', }) // DEFAULT PRODUCT ESSENCE .state('main.product.index', { url: '', views: { 'widget': { controller: 'WidgetController as PWC', templateUrl: '/routing-demo/widget.html' }, intro: { controller: IntroController as PIC, templateUrl: /routing-demo/intro.html }, content: { controller: ContentController as PCC, templateUrl: '//routing-demo/intro.html' }, content: { controller: 'ContentController as PCC', templateUrl: '//routing-demo/content.html} } } }) // SUBSTANCE PRODUCT DETAILS .state('main.product.details', { url: '/details', views: { 'widget': { controller: 'WidgetController as PWC', templateUrl: '/routing-demo/widget.html' }, 'content': { controller: 'ContentController as PCC' , templateUrl: '/routing-demo/content.html' } } }); , ['ui.router']) .config(config); That would be the first approach. What happens when I switch between main.product.index and main.product.details? Content and widget get reloaded, but we just want to reload the content. This was problematic, and developers actually created routers that would support just this feature. One of the names for it was sticky opinions. Fortunately, the ui-router supports that after eject from the box with absolute named display targeting. Comprehensive product page // without further problems .state('main.product', { abstract: true, url: ':id', views: { // TARGETING UNNAMED VIEW IN MAIN. HTML @main: { controller: 'ProductController as PC', templateUrl: '/routing-demo/product.html' }, // TARGETING TO DISPLAY A WIDGET IN THE PRODUCT. HTML // BY DEFINING THE CHILD'S VIEW, WE ALREADY ENSURE HERE THAT IT IS NOT USED TO CHANGE THE STATUS OF THE CHILD : { controller: 'WidgetController as PWC', templateUrl: '/routing-demo/widget.html' } } }) // DEFAULT PRODUCT ESSENCE .state('main.product.index', { url: '', views: { 'intro': { controller: 'IntroController as PIC', templateUrl: '/routing-demo/intro.html' }, 'content': { controller: 'ContentController as PCC', '/routing-demo/content.html' } } }) // PRODUCT DETAILS SUBSTATE .state('main.product.details', { url: '/details', views: { 'content': { { controller: as PCC, templateUrl: /routing-demo/content.html } } }); By dragging the status definition into parental view, which is also abstract, we can preserve the child's view before reloading when switching url, which usually affects that child's siblings. Of course, the widget could be a simple directive. But the point is, it could be another complex nesting condition. There is another way to do this using $urlRouterProvider.deferIntercept(), but I think using the status configuration is actually better. If you are interested in capturing the route, I wrote a small tutorial on StackOverflow. Common error #6: Declare everything in the angular world using anonymous features This error is lighter in caliber, and is more of a style issue than avoiding AngularJS error messages. You may have noticed before that I rarely hand over anonymous functions to angular internal statements. Usually I only define the function first and then pass it on. These are more than just features. I get this approach from reading style guides, especially Airbnb and Todd Motto's. I believe there are several advantages and almost no downsides to it. First, you can manipulate and mutate your functions and objects much easier if they are assigned to a variable. Secondly, the code is cleaner and can be easily divided into files. That means conservation. If you do not want to pollute the global namespace, wrap each file in IIFEs. The third reason is testability. Consider this example: use strict; function yoda() { var privateMethod = function () { // this function is not exposed }; var publicMethod1 = function () { // this function is exposed, but internal not exposed // some logic... }; var publicMethod2 = function (arg) { // LOWER CALL CANNOT BE SPIED JASMINE publicMethod1('someArgument'); // If the literal is returned in this way, can not be invoked on the inside return { publicMethod1: function () { return publicMethod1(); }, publicMethod2: function (arg) { return publicMethod2(arg); } }, } ular.module('app', []) .factory('yoda', yoda); So now we could mock the publicMethod1, but why should we do it because it is exposed? Wouldn't it be easier just to spy on an existing method? However, the method is actually a different function - thin packaging. Check out this approach: yoda() function { var privateMethod = function () { // this function is not exposed }; var publicMethod1 = function () { // this function is exposed, but it's guts are not exposed // some logic ... }; var publicMethod2 = function (arg) { // lower call can not be spied on publicMethod1 ('someArgument'); // BUT THIS CAN! hostObject.publicMethod1('aBetterArgument'); }; var hostObject= { publicMethod1: function() { return publicMethod1(); }, publicMethod2: function (arg) { return publicMethod2(arg); } }; return hostObject; } This is not just about style, because in fact the code is more reusable and idiomatic. In the event that the receives more expressive performance. Dividing the entire code into separate blocks only makes it easier. Common error #7: Doing heavy processing in angular AKA Using workers In some cases, it may be necessary to process a large number of complex objects by passing through a set of filters, decorators, and finally sorting algorithm. One use is when an app should work offline or where data display performance is key. And since JavaScript is single-threaded, it is relatively easy to freeze the browser. It's also easy to avoid it with web workers. There don't seem to be any popular libraries that handle that specifically for AngularJS. It might be the best though, because implementation is simple. First, let's set up the service: scoring functionService($q) { var scoreItems = function (items, weights) { var postponed = $q.defer(); worker var = new worker('/worker-demo/scoring.worker.js'); var orders = { items: items, scales: scales }; worker.postMessage(orders); worker.onmessage = function (e) { if (e.data & e.data.ready) { deferred.resolve(e.data.items); } }; return deferred.promise; }; var hostObject = { scoreItems: function (items, weights) { return scoreItems(items, weights); } }, return hostObject, } angular.module('app.worker') .factory('scoringService', scoringService); Now, worker: use strict; scoring function Function (items, weights) { var itemsArray = []; for (var i = 0, i < items.length; i++) { // some heavy processing // itemsArray is filled in, etc. } itemsArray.sort(function (a, b) { if (a.sum > b.sum) { return -1; } otherwise, if (a.sum < b.sum) { return 1; } else { return 0; } }); return itemsArray; } self.addEventListener('message', function (e) { var reply = { ready: true }} ; if (e.data & e.data.items & e.data.items.length) { reply.items = scoringFunction (e.data.items, e.data.items, e .data.weights); } self.postMessage(reply); }, false); Now, apply the service as usual and treat scoringService.scoreItems() as you would any method of service that returns a promise. Heavy processing is carried out on a separate thread and UX does not hem. Here's what to look out for: there doesn't seem to be a general rule on how many workers are to be put in. Some developers claim that 8 is a good number, but use the online calculator and suit yourself to check compatibility with older browsers I encountered a problem when uploading the number 0 from service to worker. I used .toString() on the handed over property, and it worked properly. Common error #8: Overusing and misunderstanding resolves adds more time to load the view. I think our primary goal is high performance front-end applications. There should be no problem rendering some parts of the view while the application waits for data from the API. Consider this setting: resolve(index, timeout) { return { data: function($q, $timeout) { var deferred = $q.defer(); $timeout(function() { resolve called ' + index)); }, }, return deferred.promise; } }; configResolves($stateProvide) function { $stateProvider // MAIN ABSTRACT STATUS, ALWAYS ON .state('main', { url: '/', controller: 'MainController as MC', templateUrl: '/routing-demo/main.html', resolve: resolve(1, 1597) }) // COMPLEX PRODUCT PAGE .state('main.product', { url: ':id', controller: 'ProductController as PC', templateUrl: '/routing-demo/product.html', resolve(2,2584) }) // PRODUCT DEFAULT SUBSTATE .state('main.product.index', { url: '', display: { 'intro': { controller: 'IntroController as PIC', template /routing-demo/intro.html }, content: { controller: ContentController as PCC, templateUrl: /routing-demo/content.html } }, resolve: resolve(3, 987) }); } The output of the console will be: Data resolve called 3 Data resolve called 1 Data resolve called 2 Main Controller executed Intro Controller executed Which basically means that: The resolves are executed asynchronously We can't rally on a order of execution (or least need to flex quite a bit) All states are blocked until all the fixes do their thing, even if they are not abstract. This means that before a user sees any output, they must wait for all dependencies. We have to have this data, sure, okay. If it is absolutely necessary to have it before displaying, insert it into the .run(). Otherwise, you just need to call the service from the controller and handle the half-loaded state gracefully. Seeing the work over - and the controller is already done, so it actually is progress - is better than having an app stand. Common Error #9: Not Optimization App – Three examples a) Causes too many digest loops such as connecting sliders to models This is a general problem that may result in AngularJS errors, but I will discuss this on the example sliders. I used this slider library, an angular range slider, because I needed advanced functionality. This Directive has the following syntax in the minimum version: <body ng-controller=MainController as MC><div range-slider= min=0 max=MC.maxPrice pin-handle=min model-max=MC.price></div></points> Consider the following code in the controller: this.maxPrice = '100'; this.price = 55; $scope.$watch('MC.price', function (newVal) { if (newVal || newVal === 0) { for (var i = 0, i < 987;= i++)= {= console.log('all= your= base= are= belong= to= us');= set= a= timeout= on= the= input.= but= that= is= not= always= handy,= and= sometimes= we= don't= really= want= to= delay= the= actual= model= change= in= all= cases.= so= we= will= add= a= temporary= model= bound= to= change= the= working= model= on= timeout:=><body ng-controller=MainController as MC><div range-slider= min=0 max=MC.maxPrice pin-handle=min model-max=MC.priceTemporary></div></points> and in the driver: this.maxPrice = 100; this.price = 55; toto.priceTemporary = 55; $scope.$watch('MC.price', function (newVal) { if (!isNaN(newVal)) { for (var i = 0, i < 987; i++) 987;= i++)=></ i++) > > console.log (ALL YOUR BASES BELONG TO US); } } }); var timeoutInstance; $scope.$watch('MC.priceTemporary', function (newVal) { if (!isNaN(newVal)) { if (timeoutInstance) { $timeout.cancel(timeoutInstance); } timeoutInstance = $timeout(function() { $scope. MC.price = newVal; }, 144); } }); b) Does not use $applyAsync AngularJS does not have an electoral mechanism to call $digest(). It is done only because we use guidelines (e.g. NG click, input), services ($timeout, $http) and methods ($watch) that evaluate our code and then call digest. What .$applyAsync() does is delay the resolution of expressions until the next $digest() cycle, which runs after a 0 timeout, which actually is ~10ms. There are two ways to use applyAsync now. Automated method $http applications and manual method for others. If you want all http requests that are returned around the same time to resolve in one digest, do: mymodule.config (function ($httpProvider) { $httpProvider.useApplyAsync(true); }); The manual way shows how it actually works. Consider some feature that runs on callback to a vanilla JS event listener or jQuery .click(), or some other external library. After you start and change models, if you have not yet packaged it in $apply() you need to call $scope.$root.$digest() ($rootScope.$digest()), or at least $scope.$digest(). Otherwise, nothing will change. If you do this multiple times in one procedure, it may start working slowly. Consider calling $scope.$applyAsync() to terms instead. Sets only one digest cycle for all of them. c) Do heavy image processing If you experience poor performance, you can find out the reason using a timeline from Chrome Developer Tools. I will write more about this tool by mistake #17. If a timeline chart dominates with green after uploading, performance issues may be related to image processing. It is not strictly related to AngularJS, but can happen on top of AngularJS performance problems (which would be mostly yellow on the chart). As constructors, we need to think about the complete end of the project. Take a moment to assess: Are you using paralaxes? Do you have several layers of content overlapping each other? Moving your pictures? Image scale (e.g. background size)? Do you resize images in loops, and can cause you to spend loops on resizing? If you answered yes to at least three of the above, consider mitigating it. Maybe you can serve different image sizes and not resize at all. Maybe you could add a transformation: translateZ (0) force GPU processing hack. Or, use the requestAnimationFrame for manipulators. Common error #10: jQuerying It – Separated BY DOM Tree Many times you probably hear that it is not recommended to use jQuery with AngularJS, and that it should be avoided. It is essential to understand the reason for these statements. There are at least three reasons as far as I can see, but none of them are real blockers. 1: When you run the jQuery code, you must $digest () yourself. In many cases, there is an AngularJS solution that is customized for AngularJS and may be better used inside an angular than jQuery (e.g. ng-click or event system). Reason 2: A way of thinking about building an app. If you've been adding JavaScript to webpages that reload when navigating, you don't have to worry too much about memory usage. You'll have to worry about one-page apps. If you don't clean up, users who spend more than a few minutes in your app may experience performance-enhancing problems. Reason 3: Cleaning up isn't actually the easiest thing to do and analyze. There is no way to call a garbage can from the script (in the browser). You may end up with separate DOM trees. I created an example (jQuery is loaded in index.html): <section> <test-for-toptal></test-for-toptal> <button ng-click=MC.removeDirective()>remove function</button> </section> MainController($rootScope, $scope) { this.removeDirective = function () { $rootScope.$emit('destroyDirective'); }; } Test functionForToptal($rootScope, $timeout) { return { link: function (range, element, attributes) { var destroyListener = $rootScope.$on('destroyDirective', function () { scope.$destroy(), }); // add timeout for DOM to prepare $timeout(function() { scope.toBeDetached = element.find('p'); }); scope.$on('$destroy', function() { destroyListener(); element.remove(); }); }, template: '<div><p>SOM DIRECTIVE</p></div>' }; } ular.module('app', []) .controller('MainController', MainController) .directive('testForToptal', testForToptal); This is a simple directive that will make a text. Below it is a button that only manually destroys the directive. So when the directive is removed, the reference to the DOM tree remains in the range.toBeDetached. In chrome dev tools, if you have access to bookmark profiles and then take a heap shot, you will see in the output: You can live with a few, but it's bad if you have a ton. Especially if for some reason, as in the example, you can save to a range. The entire DOM will be evaluated on each digest. The problematic detached DOM tree is a tree with 4 nodes. So how can this be solved? scope.$on('$destroy', function () { // setting this model to null // resolves the issue. scope.toBeDetached = null; destroyListener(); element.remove(); Separate DOM tree with 4 items will be removed! In this example, the Directive uses the same scope and imposes the DOM element within the scope. It was easier for me to prove it that way. This is not always as bad as you could save into a variable. However, it would still occupy memory if any foreclosure that referenced that variable or any other of the same function range lived on. Common #11: Overuse of an isolated range Whenever you need a directive that you know will be used in one place or that you do not expect to conflict with used in, it is not necessary to use an isolated range. Recently, there is a trend to create reusable components, but did you know that basic angular guidelines do not use an isolated range at all? There are two main reasons: you cannot apply two isolated scope directives to an element, and you may encounter problems with nesting/heredity/event processing. Especially when it comes to transclusion – effects may not be what you expect. So it would fail: <p isolated-scope-directive= another-isolated-scope-directive= ng-if=MC.quux ng-repeat=q in MC.quux> And even if you use only one directive, you will notice that neither isolated range models nor events broadcast in isolatedScopeDirective will be available to AnotherController. This is sad, you can flex and use transclusion magic to make it work – but for most use cases, there is no need to isolate it. <p isolated-scope-directive= ng-if=MC.quux ng-repeat=q in MC.quux><div ng-controller=AnotherController> ... isolated scope is not available here, see: {{ IsolatedModel }} </div></p> So two questions now: How can you process superior scope models in a directive with the same scope? How can you instantiate new model values? There are two ways in which you pass values to attributes in both of them. Consider this MainController: MainController($interval) function { this.foo = { bar: 1 }; this.baz = 1; var that = this; $interval(function () { that.foo.bar++; }, 144); $interval(function () { that.baz++; }, 144); this.quux = [1,2,3]; } This controls this view: <body ng-controller=MainController as MC><div class=cyan-surface><h1 style=font-size: 21px;>Attribute Test</h1><test-directive watch-attribute=MC.foo observe-attribute=current index: {{ MC.baz }}></test-directive></div></body> Note that the watch attribute is not interpolated. It all works, thanks to the magic of JS. Here is the definition of the directive:

testDirective() function { var postLink = function (range, element, attrs) { scope.$watch(attrs.watchAttribute, function (newVal) { if (newVal) { // look in the console // we can not use the attribute directly console.log (attrs.watchAttribute); // newVal is evaluated, and can be used scope.modifiedFooBar = newVal.bar * 10; } }, true); attrs.$observe('observeAttribute', function (newVal) { scope.observed = newVal; }); }; return { link: postLink, templateUrl: '/attributes-demo/test-directive.html' }; } Note that attrs.watchAttribute is uploaded to scope.$watch() without quotes! That is, what was known $watch was a string of MC.foo! However, this works because any string passed to $watch() gets evaluated by scope and MC.foo is available on the range. This is also the most common way that attributes are tracked in the basic AngularJS guidelines. Check out the code on github for the template, and look into $parse and $eval for even more awesomeness. Common error #12: No cleaning up in a row - intervals, time limits and variables AngularJS AngularJS some work on your behalf, but not all. The following needs to be manually cleaned: All observers who are not bound by the current range (e.g. bound to $rootScope) Intervals Timeouts Variables referring to DOM in directives Dodgy jQuery plugins, such as those that do not have manipulators to respond to a JavaScript event $destroy If you do not want to do this manually, you will encounter unexpected behavior and memory leaking. Even worse - these will not be immediately visible, but they will creep up eventually. Murphy's Law. The amazing thing is that AngularJS provides useful ways to how to deal with all of these: cleanMeUp ($interval, $rootScope, $timeout) { var postLink = function (range, element, attrs) { var rootModelListener = $rootScope.$watch('someModel', function () { // do something }); var myInterval = $interval(function () { // do something at intervals }, 2584); var myTimeout = $timeout(function() { // postpone any action here }, 1597); scope.domElement = element; $timeout(function) { // call $destroy manually for testing purposes scope.$destroy(); }, 987); // here is where the cleanup happens scope.$on('$destroy', function () { // disable listener rootModelListener(); // cancel interval and timeout $interval.cancel (myInterval); $timeout.cancel (myTimeoutout).); // cancel DOM-bound model scope.domElement = null; }); element.on ('$destroy', function () { // this is a jQuery event // clean up all vanilla JavaScript / jQuery artifacts here // respectful jQuery plugins have $destroy manipulators, // that's why this event is emitted ... // follow the standards. }); Note the jQuery $destroy event. It is called as an AngularJS one, but it is handled separately. The $watchers will not respond to the jQuery event. Common mistake #13: Keeping too many observers It should be pretty easy now. There's one thing to understand here: $digest (). For each binding {{ model }}, AngularJS creates an observer. For each stage of digestion, each such bond shall be evaluated and compared with the previous value. It's called dirty control, and that's $digest does. If the value has changed since the last check, the observer's callback shall be thrown out. If this observer callback modifies the model ($scope variable), a new cycle of data (up to 10) is $digest thrown out when the exemption is thrown out. Browsers don't even have problems with thousands of links unless the terms are complex. The common answer to how many observers it is ok to have is 2000. So, how can we limit the number of observers? By tracking the range of models when we don't expect it to change. It's fairly easy further away from AngularJS 1.3 because one-time ties are at the core now. <if ng-repeat=item in ::vastArray>{{ ::item.speed }} Once vastArray and item.velocity have been evaluated, they will never change again. You can still apply filters to the field, they will work just fine. It's just that the field itself is not evaluated. In many cases, it's a win. error #14: Misunderstanding Misunderstanding Digest This AngularJS error was already partially covered by errors 9.b and 13. This is a more thorough explanation. AngularJS updates the DOM due to the callback function for observers. Every binding, that's the directive {{ someModel }} sets the observers, but observers are also set for many other directives such as ng-if and ng-repeat. Just look at the source code, it's very readable. Observers can also be set up manually, and you've probably done that at least a few times yourself. $watch()ers are bound to ranges. $Watchers may take strings that are evaluated to the extent that $watch() has been bound. They can also evaluate features. And they also take callbacks. So, when $rootScope.$digest() is called, all registered models (i.e. $scope variables) are evaluated and compared with their previous values. If the values do not match, a callback of the $watch() shall be made. It is important to understand that even if the model value has been changed, callback does not fire until the next stage of digestion. This is called a phase for some reason - it can consist of several digestive cycles. If only the observer changes the range model, a further digestion cycle is performed. But $digest() not interviewed. It is called basic directives, services, methods, etc. If you change a model from a custom function that does not name .$apply, .$applyAsync, .$evalAsync, or anything else that eventually calls $digest(), the bindings are not updated. By the way, the source code $digest() is actually quite complex. Still, it's worth reading because cheerful warnings have quantified it. Common error #15: Don't rely on automation or rely on it too much if you follow trends within front-end developments and are a little lazy – like me – then you probably try to do everything manually. Tracking all your dependencies, processing files in different ways, reloading your browser after each file save – there is much more to develop than just encoding. So you may be using bower, and maybe npm depending on how you serve your application. There is a chance that you may be using a grunt, sip, or brunch. Or bash, which is also cool. In fact, you may have started your latest project with some Yeoman generator! This leads to the question: do you understand the whole process of what your infrastructure really does? Need what you have, especially if you've just spent hours trying to fix the webserver livereload connection feature? Take a second to assess what you need. All these tools are here just to help you, there is no other reward for using them. The more experienced developers I speak to tend to simplify things. Common error #16: Unlaunched drive tests in TDD mode tests will not be your code without AngularJS error messages. What they do is make sure your team doesn't run into regression problems all the time. I am writing specifically about unit tests here, not because I feel they are more important than e2e tests, because they perform much faster. I must confess that the process I am about to describe is very pleasant. Test Driven Development as an implementation for eg sip-karma runner, basically running all your drive tests on each file to save. My favorite way to write tests is, I just write blank assurances first: describe ('some module', function () { it ('should call name-it service ...', function () { // leave it blank now }); ... }); After that, I write or refactor the actual code, then I go back to the tests and fill out the warranties with the actual test code. Having a TDD job running in a terminal speeds up the process by about 100%. Unit tests are performed in seconds, even if you have many. Just save the test file and the runner will pick up, evaluate your tests, and provide feedback immediately. With e2e tests, the process is much slower. My advice - divide the e2e tests into test suites and just run them one by one. Protractor has support for them, and below is the code that I use for my test tasks (I like a sip). use strictly; boil sip = require('sip'); var args = require('yarn').argv; var browserSync = require('browser-sync'); var karma = require('gulp-karma'); var protractor = require('gulp-protractor').retread; var webdriverUpdate = require('gulp-protractor').webdriver_update; function test() { // Be sure to return the stream // NOTE: Using the fake './foobar' so that in order to run the files // listed in karma.conf.js instead of what was uploaded to // gulp.src ! return gulp.src ('./foobar') .pipe(karma({ configFile: 'test/karma.conf.js', action: run })) .on('error', function(err) { // Make sure that failed tests cause, that you terminate the sip of the non-zero console // console.log(err); toto.emit('end'); //instead of a stream error, end it }); } function tdd() { return gulp.src('./foobar') .pipe(karma({ configFile: 'test/karma.conf.js', action: 'start' })) .on('error', function(e r) { // Make sure that unsuccessful tests cause the sip to exit the non-zero // console.log(err); // this.emit('end'); // does not end the current here }); } runProtractor () { var argument = args.suite || everything; NOTE: Using the fake './foobar' so as to run the files // listed in protractor.conf.js, instead of what was passed // gulp.src return gulp.src('./foobar') .pipe(protractor({ configFile: 'test/protractor.conf.js', args: ['-suite', argument] })) .on('error', function (err) { // Make sure that the failed tests cause the sip to end the non-zero throw error; }) .on('end', function () { // Close browsersync server browserSync.exit(); } gulp.task('tdd', tdd); gulp.task(test, test); gulp.task('test-e2e', ['webdriver-update'], runProtractor); gulp.task('webdriver-update', webdriverUpdate); A - Chrome breakpoints Chrome dev tools allow you to point to a specific location in any of the files loaded into the browser, pause code execution at that point, and allow you to communicate with variables available from this point on. That's a lot! This feature does not require the addition of a code Everything, everything happens in dev tools. Not only do you have access to all variables, you can also see the call stack, print tray tracks, and more. You can even configure it to work with minified files. Read about it here. There are other ways you can get similar access to running, such as adding .log(). But the border points are more sophisticated. AngularJS also allows you to access the scope through DOM elements (if debuggingInfo is enabled) and apply available services through the console. Consider the following in the console: $(document.points).scope().$root or point to an element in the inspector. and then: $($0).scope() Even if debugInfo is not enabled, you can do it: ular.reloadWithDebugInfo() And have it available after reloading: To apply and communicate with the service from the console, try: var injector = $(document.body).injector(); var someService = injector.get('someService'); B - Chrome Timeline Another great tool that comes with dev tools is timeline. This allows you to record and analyze your app's live performance while you're using it. Output shows, among other things, memory usage, frame rate, and dissection of various processes that take up the processor: loading, scripting, rendering, and painting. If you experience that the performance of your app is decreasing, you will most likely be able to find the cause through that timeline tab. Just record your actions that led to performance problems and see what happens. Too many observers? You will see yellow stripes taking up a lot of space. Memory leak? In the chart, you can see how much memory has been consumed over time. Step-by-step description: C - Remote app review on iOS and Android If you're developing a hybrid app or responsive web app, you can access your device's console, DOM tree, and all other tools available through either Chrome or Safari developer tools. This includes WebView and UIWebView. First, start the Web server on the host computer 0.0.0.0 so that it can be accessed from the local network. Enable web inspector in settings. Then connect the device to the desktop and access the local development page using an ip computer instead of the normal localhost. That's all it takes, your device should now be available from your computer's browser. Here are step-by-step instructions for Android A for iOS, unofficial tutorials can be easily found through Google. I recently had some cool experience with browserSync. It works in a similar way to livereload, but also actually syncs all browsers that are viewing on the same page via browserSync. This includes user interaction such as scrolling, clicking buttons, etc. I was looking at the iOS app log output while checking the page for an iPad from my desktop. It worked nicely! Common error #18: No reading code on ng-init example ng-init, from sound, should be similar to ng-if and ng-repeat, right? Have you ever wondered why there is in documents that should not be used? IMHO it was surprising! I would expect the directive to initiate a model. This is also what he does, but ... implemented in a different way, i.e. it does not look at the attribute value. No need to browse AngularJS source code - let me tell you this: var ngInitDirective = ngDirective (priority: 450, compilation: function() { return { for: function (range, element, attrs) { scope.$eval(attrs.ngInit); } }; }); Less than you'd expect? Quite readable, apart from the embarrassing syntax of the directive, right? The sixth row is what it is all about. Compare it to ng-show: var ngShowDirective = ['$animate', function($animate) { return { restrict: 'A', multiElement: true, reference: function (range, element, attr) { scope.$watch(attr.ngShow, function ngShowWatchAction(value) { // we add a temporary, animation-specific class for ng-hide, because in this way // we can control when the element is actually displayed on the screen without having to // have a global/greedy CSS selector that breaks when other animations are run down. // Read: $animate [value? removeClass : addClass](element, NG_HIDE_CLASS, { tempClasses: NG_HIDE_IN_PROGRESS_CLASS }); }); } }; }]; Again, the sixth row. There is $watch, that is what makes this directive dynamic. In the AngularJS source code, a large part of all codes are comments that describe code that has been mostly readable from the beginning. I believe this is a great way to learn about AngularJS. Conclusion This guide applies to the most common errors AngularJS is almost twice as long as other tutorials. It turned out so naturally. The demand for high-quality javascript edi-queuing engineers is very high. AngularJS is so hot right now and it has been holding a stable position among the most popular development tools for several years. With AngularJS 2.0 on the way, it will likely dominate for years to come. What's great about front-end development is that it is very rewarding. Our work is visible immediately and people interact directly with the products we supply. Time spent learning JavaScript, and I believe we should focus on JavaScript, is a very good investment. It's the language of the Internet. The competition is super strong! There's one focus for us – the user experience. To be successful, we have to cover everything. You can download the source code used in these examples from GitHub. Feel free to download it and make it your own. I wanted to give credits to the four publishing developers that inspired me the most: Ben Nadel Todd Motto Pascal Precht Sandeep Panda I also wanted to thank all the great people at FreeNode #angularjs and #javascript channels for many excellent interviews, and continuous support. Finally, always remember: // when in doubt, comment it! :) :)