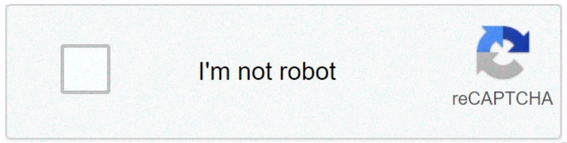

Continue

Angularjs templateurl cors

Show / Hide Table of Contents An error occurred while loading this resource. Please try again later. Over the past few months I've been looking at ways to improve runtime performance at The Iant SPA that I'm working on at Domo. We've made some serious progress, but with a million lines of code in a SPA, some of the changes aren't always easy. One of our team members made the discovery to add lazyloading to AngularJS projects, and we have invested heavily in this. A few different team members (Jason and Tim) dove to help us measure the time it takes our app to fully initialize. We have also used webpacks to streamline the construction, as well as change some of the patterns that we use. When combining webpack with ocLazyload, we have found a serious profit for AngularJS projects. This last week I took on the task of changing all the component/directive template declarations, and changing them from templateUrl to template. Instead of manually moving all templates from their separate .html files to their respective JS files, we decided to use a webpack-in loader, and require the templates to be inline strings. To better explain it... Let me show you what I mean. The following is a sample AngularJS component:As you can see, in the first example there is a component that uses a templateUrl to load the template. This is problematic at best, IMO. This means that you will need to either deploy the foo/bar/myComponent.html file to production so that your production app can load the template fragment via a second network request to get it, OR it means you will need to add a build step that will find all instances of templateUrl and bring these templates into the AngularJS templateCache. Both of these solutions have problems. The problems with the first are obvious: if all your templates in production required a separate network request to get them, then loading any single view would require N network requests to get all the feedback, where N is the number of components/directives/ngIncludes in your opinion. The problem with the second is that the building steps, while super handy, will load all your templates into your main webpack bundle. This means that even when you think lazyload a component, or an entire section of components, their templates will still be loaded with your main stack. So, you can't take full advantage of the benefits you get from lazyloading. Given the many hundreds and hundreds of templates that we have in our project, none of these were feasible. We needed something else. We needed something that would allow us to load our templates efficiently, without separate network requests for each one, while also allowing us to lazyload the same templates. So we decided to look at using a webpack loader that would allow us to require our templates in our components like inline strings of HTML/Angular templates. The BenefitsBy uses webpack to load .html files, we we that we were able to efficiently load our templates, while also allowing us to fully benefit from lazyloading. When you use the template: require('foo/bar/my.html') syntax, the webpack replaces your require statement with a function that is called and returned with the string for the template. Since the template is now provided as an html string, if you lazyload the component, the template will also be lazyloaded. This is exactly what we needed. However, we discovered several other benefits, the discovery of which prompted this post. Faster component initialization — When you use an inline string as a template, the component can initialize synchronously. By using templateUrl, AngularJS will request the template from the Cache template. Because templateCache may already have the template in it's cache, or may need to go to over the network to get it, requesting a template from the cache is a process that happens asynchronously. Even if the template already exists in the cache, templateCache will return the already cached template via a promising-based call. This means that the component cannot initialize in the same event loop. Requests to templateCache will always be placed on the next event loop, even in the best of scenarios. This means that the component can start initializing, request it to be the template, and then end initialization in the next event loop. However, when you use an inline string, the component already has it, the template is ready, so it can start and end it's initialization in the same event loop. This may not seem significant, but it had several unexpected results that we had to compensate for. - Components initialize faster — which sounds awesome, AIR? Well, that's awesome. However, this means that some of your components that have always had their input values defined when their initialize can break, cause the same values may not be there yet. We had multiple component break, due to undefined input binding values. We had to change these components to $watch $onChanges or $onChanges to detect the update to the input values. - Unit tests will run differently — Since write test changes when you do a synchronous test vs. an asynchronous test, the test for these components can definitely be changed. For example, in Mocha, if your test is async, you inject the method done in your test, and call it when the test is done. We found that the tests were now performed synchronously, which meant that the need to inject was no longer necessary. Furthermore, and it's embarrassing to admit this, but we had tests that were written synchronously, but with the templates being async, these tests had never been successfully completed. So when I committed the changes to the inline templates, these tests began to successfully run, and instead of passing, they were failing!!!! At first I thought I had broken all these tests. It wasn't until after 5 hours of poking around that I realized that these tests were never passing. So we actually has increased test coverage now that we use inline templates.html-loader uses an html-minifier — This small fact instantly reduced the size of our templates by 19% across the app. This is so outstanding, and is certainly something that we should have done for a long time. It also parsed the templates, and helped us find a few dozen templates that had invalid html in them. Things like: classy, where = was missing. Or attribute={{something}} , which lacks the quotes around everything. When I fixed them, the build worked again.ng-includes were still broken — While the component templates were now working, ng-include's were broken now. We needed to do something for them. So we built a small custom loader, which will bring the template into the templateCache. Our internal practices tell us not to use ng-include, but we still have lots of 3+ year old code containing them. So, rather than refactor all this into this commit, I used this new loader, and went to every section of the app that has a ng-include and loaded template for that section, as I've shown below. This means that ng-include's are also taken care of in this new process. Used JSCodeShiftI fully recommends using webpacks for AngularJS apps, and using html loader to get your templates inline, which means you will need to change your templateUrl instances to template instances. Since they all look very different, I decided that this was a very good uses case for JSCodeShift, a project from Facebook that lets you crawl AST, and programmatically replace all instances for you. You can think of it as Find & Replace on Steroids, Injected w/ More Steroids. It was really easy to write the script that found and updated all these uses of templateUrl: 'some/url/to.html with template: require(). I was able to change 90% of the uses programmatically (about 700 files), and I had to finish the last 70 by hand. I could have written the code to finish the other 70, but I figured I could make them easier by hand than by trying to encode them each individually. A quick note, ast Explorer is an absolute must when using JSCodeShift. Without it, I would not have been able to make any progress. ConclusionGet your AngularJS apps on a webpack build, and spend time getting them over to using html loader to load your templates. Use template instead of templateUrl, and if you haven't already done so, stop using ng-include. And then, lazyload, lazyload, lazyload! Some times people distinguish between delayed loading and lazy loading. I refer to both delayed loading and lazy loading when I say lazyload. It's your best change on reducing time to First Meaningful Paint, and reducing the time to having an app that the user can interact with. Back on your heads! Look at 4k Star 59.5k Fork 28.6k You can't perform that action right now. You signed in with a different tab or window. to update your session. You You in another tab or window. Reload to update your session. We use optional third-party analytics cookies to understand how you use GitHub.com so we can build better products. Learn more. We use optional third-party analytics cookies to understand how you use GitHub.com so we can build better products. You can always update your selection by clicking cookie preferences at the bottom of the page. For more information, see our Privacy Statement. We use important cookies to perform important website functions, such as the internet. Read more Always active We use analytics cookies to understand how you use our websites so that we can make them better, e.g. Read more