I'm not robot
reCAPTCHA
Continue

Javascript get domain name without subdomain

I’m sure this is a common problem: getting the domain or host name from a URL using JavaScript. There are certainly many solutions to this problem out there. However, the solutions I found weren’t robust enough for my needs, so I ended up writing my own. The Problem With Most Solutions Most of the solutions I found suffered from one or more of the following problems: They rely on window.location.hostname (or similar) The vast majority of solutions out there work by parsing window.location.hostname or window.location.href. This would be fine if I was working out the domain of the page I’m currently on, but I want to work it out for a URL that I’m not actually visiting at the moment. It has to work for any URL stored in a variable in string format. They don’t cater for URL parameters or hash URLs Most solutions parse the URL based only on the forward slash. That works for the most common URLs, such as which will return scratch99.com, but it won’t work with the following URLs: #footer scratch99.com?s=web+development which would return the full URL in both cases. I need it to strip out everything after the ? and # characters. They don’t strip out the http:// I’m processing the URL so I can store the domain and use it in a URL, in the following way: I do not want the following: //theparseddomain.com/ so I need to strip out the http:// when parsing the URL. How To Parse A URL And Get The Hostname My solution, taking all of the problems above into account, is as follows: [sourcecode language=“javascript”] var url = " ; var domain = url.replace(‘http://’,”).replace(‘https://’,”).split(/[/?#]/)[0]; [/sourcecode] This feature is no longer recommended. Though some browsers might still support it, it may have already been removed from the relevant web standards, may be in the process of being dropped, or may only be kept for compatibility purposes. Avoid using it, and update existing code if possible; see the compatibility table at the bottom of this page to guide your decision. Be aware that this feature may cease to work at any time. The domain property of the Document interface gets/sets the domain portion of the origin of the current document, as used by the same-origin policy.const domainString = document.domain The getter for this property returns the domain portion of the current document's origin. In most cases, this will be the hostname portion of the document's URL. However, there are some exceptions: If the page has an opaque origin, e.g. for a page with a data URL, then it will return the empty string. If the document.domain setter has been used, then it will return the value that was set. Usually it is better to use the Location.hostname property instead.document.domain = domainString The setter for this property can be used to change a page's origin, and thus modify how certain security checks are performed. It can only be set to the same or a parent domain. For example, if and both use document.domain = "example.com"; then they have both modified their origin to have the same domain, and they can now access each other's DOM directly—despite being cross-origin, which would normally prevent such access. Note that setting document.domain to its current value is not a no-op. It still changes the origin. For example, if one page sets document.domain = document.domain; then it will be counted as cross-origin from any other normally-same-origin pages that have not done the same thing. Deprecation The document.domain setter is deprecated. It undermines the security protections provided by the same origin policy, and complicates the origin model in browsers, leading to interoperability problems and security bugs. Attempting to set document.domain is dangerous. It opens up full access to a page's DOM from all subdomains, which is likely not be what is intended. It also removes the port component from the origin, so now your page can be accessed by other pages with the same IP address or same host component, even on a different port. This is especially insecure on shared hosting. For example, another shared hosting customer is able to host a site at the same IP address but on a different port, then setting document.domain will remove the same-origin protection that normally protects you from that other customer's site accessing your site's data. Similar problems occur with shared hosting sites that give each customer a different subdomain. If a site sets document.domain, any other customer on a different subdomain can now do the same thing, and start accessing the data of the original site. Instead of using document.domain to facilitate cross-origin communication, you should use Window.postMessage to send an asynchronous message to the other origin. This controlled access via message-passing is much more secure than the blanket exposure of all data caused by document.domain. Failures The setter will throw a "SecurityError" DOMException in several cases: As an example of this last failure case, trying to set document.domain to "example.org" when on will throw. Additionally, as part of its deprecation, it will do nothing when combined with certain modern isolation features: Finally, setting document.domain does not change the origin used for origin-checks by some Web APIs, preventing sub-domain access via this mechanism. Affected APIs include (but are not limited to): Window.localStorage, IndexedDB_API, BroadcastChannel, SharedWorker . For code running at the URL this example would set currentDomain to the string "developer.mozilla.org". const currentDomain = document.domain; Although the getter is not dangerous in the same way that the setter is, it is likely simpler and more useful to use the Location.hostname property instead. Then you can avoid document.domain entirely: const currentHostname = location.hostname; For the URL currentHostname is also the string "developer.mozilla.org". Other alternatives that provide slightly different information are Location.host, which includes the port, and WindowOrWorkerGlobalScope.origin, which provides the full origin.BCD tables only load in the browser The same-origin policy is a critical security mechanism that restricts how a document or script loaded by one origin can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors. For example, it prevents a malicious website on the Internet from running JS in a browser to read data from a third-party webmail service (which the user is signed into) or a company intranet (which is protected from direct access by the attacker by not having a public IP address) and relaying that data to the attacker.Two URLs have the same origin if the protocol, port (if specified), and host are the same for both. You may see this referenced as the "scheme/host/port tuple", or just "tuple". (A "tuple" is a set of items that together comprise a whole — a generic form for double/triple/quadruple/quintuple/etc.) The following table gives examples of origin comparisons with the URL URL Outcome Reason Same origin Only the path differs Same origin Only the path differs Failure Different protocol Failure Different port (http:// is port 80 by default) Failure Different host Scripts executed from pages with an about:blank or javascript: URL inherit the origin of the document containing that URL, since these types of URLs do not contain information about an origin server. For example, about:blank is often used as a URL of new, empty popup windows into which the parent script writes content (e.g. via the Window.open() mechanism). If this popup also contains JavaScript, that script would inherit the same origin as the script that created it. data: URLs get a new, empty, security context. Internet Explorer has two major exceptions to the same-origin policy: Trust Zones If both domains are in the highly trusted zone (e.g. corporate intranet domains), then the same-origin limitations are not applied. Port IE doesn't include port into same-origin checks. Therefore, and are considered the same origin and no restrictions are applied. These exceptions are nonstandard and unsupported in any other browser. The approach described here (using the document.domain setter) is deprecated because it undermines the security protections provided by the same origin policy, and complicates the origin model in browsers, leading to interoperability problems and security bugs. A page may change its own origin, with some limitations. A script can set the value of document.domain to its current domain or a superdomain of its current domain. If set to a superdomain of the current domain, the shorter superdomain is used for same-origin checks. For example, assume a script from the document at executes the following: document.domain = "company.com"; Afterward, the page can pass the same-origin check with (assuming sets its document.domain to "company.com" to indicate that it wishes to allow that - see document.domain for more). However, company.com could not set document.domain to othercompany.com, since that is not a superdomain of company.com. The port number is checked separately by the browser. Any call to document.domain, including document.domain = document.domain, causes the port number to be overwritten with null. Therefore, one cannot make company.com:8080 talk to company.com by only setting document.domain = "company.com" in the first. It has to be set in both so their port numbers are both null. The mechanism has some limitations. For example, it will throw a "SecurityError" DOMException if the document-domain Feature-Policy is enabled or the document is in a sandboxed , and changing the origin in this way does not affect the origin checks used by many Web APIs (e.g. localStorage, indexedDB, BroadcastChannel, SharedWorker). A more exhaustive list of failure cases can be found in Document.domain > Failures. Note: When using document.domain to allow a subdomain to access its parent, you need to set document.domain to the same value in both the parent domain and the subdomain. This is necessary even if doing so is setting the parent domain back to its original value. Failure to do this may result in permission errors. The same-origin policy controls interactions between two different origins, such as when you use XMLHttpRequest or an element. These interactions are typically placed into three categories: Cross-origin writes are typically allowed. Examples are links, redirects, and form submissions. Some HTTP requests require preflight. Cross-origin embedding is typically allowed. (Examples are listed below.) Cross-origin reads are typically disallowed, but read access is often leaked by embedding. For example, you can read the dimensions of an embedded image, the actions of an embedded script, or the availability of an embedded resource. Here are some examples of resources which may be embedded cross-origin: JavaScript with . Error details for syntax errors are only available for same-origin scripts. CSS applied with . Due to the relaxed syntax rules of CSS, cross-origin CSS requires a correct Content-Type header. Restrictions vary by browser: Internet Explorer, Firefox, Chrome , Safari (scroll down to CVE-2010-0051) and Opera. Images displayed by . Media played by and . External resources embedded with and . Fonts applied with @font-face. Some browsers allow cross-origin fonts, others require same-origin. Anything embedded by . Sites can use the X-Frame-Options header to prevent cross-origin framing. Use CORS to allow cross-origin access. CORS is a part of HTTP that lets servers specify any other hosts from which a browser should permit loading of content. To prevent cross-origin writes, check an unguessable token in the request — known as a Cross-Site Request Forgery (CSRF) token. You must prevent cross-origin reads of pages that require this token. To prevent cross-origin reads of a resource, ensure that it is not embeddable. It is often necessary to prevent embedding because embedding a resource always leaks some information about it. To prevent cross-origin embeds, ensure that your resource cannot be interpreted as one of the embeddable formats listed above. Browsers may not respect the Content-Type header. For example, if you point a tag at an HTML document, the browser will try to parse the HTML as JavaScript. When your resource is not an entry point to your site, you can also use a CSRF token to prevent embedding. The following cross-origin access to Location properties is allowed: Attributes URLUtils.href Write-only. Some browsers allow access to more properties than the above.Access to data stored in the browser such as Web Storage and IndexedDB are separated by origin. Each origin gets its own separate storage, and JavaScript in one origin cannot read from or write to the storage belonging to another origin. Cookies use a separate definition of origins. A page can set a cookie for its own domain or any parent domain, as long as the parent domain is not a public suffix. Firefox and Chrome use the Public Suffix List to determine if a domain is a public suffix. Internet Explorer uses its own internal method to determine if a domain is a public suffix. The browser will make a cookie available to the given domain including any sub-domains, no matter which protocol (HTTP/HTTPS) or port is used. When you set a cookie, you can limit its availability using the Domain, Path, Secure, and HttpOnly flags. When you read a cookie, you cannot see from where it was set. Even if you use only secure https connections, any cookie you see may have been set using an insecure connection. Author(s): Jesse Ruderman Custom domain names are simpler and more intuitive URLs that you can provide to your API users. After deploying your API, you (and your customers) can invoke the API using the default base URL of the following format: api-id is generated by API Gateway, region (AWS Region) is specified by you when creating the API, and stage is specified by you when deploying the API. The hostname portion of the URL (that is, api-id.execute-api.region.amazonaws.com) refers to an API endpoint. The default API endpoint can be difficult to recall and not user-friendly. With custom domain names, you can set up your API's hostname, and choose a base path (for example, myservice) to map the alternative URL to your API. For example, a more user-friendly API base URL can become: If you don't set any base mapping under a custom domain name, the resulting API's base URL is the same as the custom domain (for example,). In this case, the custom domain name can't support more than one API. A Regional custom domain can be associated with REST APIs and HTTP APIs. You can use API Gateway Version 2 APIs to create and manage Regional custom domain names for REST APIs. Custom domain names are not supported for private APIs. You can choose a minimum TLS version that your REST API supports. For REST APIs, you can choose TLS 1.2 or TLS 1.0. Register a domain name You must have a registered internet domain name in order to set up custom domain names for your APIs. If needed, you can register an internet domain using Amazon Route 53 or using a third-party domain registrar of your choice. An API's custom domain name can be the name of a subdomain or the root domain (also known as "zone apex") of a registered internet domain. After a custom domain name is created in API Gateway, you must create or update your DNS provider's resource record to map to your API endpoint. Without such a mapping, API requests bound for the custom domain name cannot reach API Gateway. An edge-optimized custom domain name is created in a specific Region and owned by a specific AWS account. Moving such a custom domain name between Regions or AWS accounts involves deleting the existing CloudFront distribution and creating a new one. The process may take approximately 30 minutes before the new custom domain name becomes available. For more information, see Updating CloudFront Distributions. Edge-optimized custom domain names When you deploy an edge-optimized API, API Gateway sets up an Amazon CloudFront distribution and a DNS record to map the API domain name to the CloudFront distribution domain name. Requests for the API are then routed to API Gateway through the mapped CloudFront distribution. When you create a custom domain name for an edge-optimized API, API Gateway sets up a CloudFront distribution. But you must set up a DNS record to map the custom domain name to the CloudFront distribution domain name. This mapping is for API requests that are bound for the custom domain name to be routed to API Gateway through the mapped CloudFront distribution. You must also provide a certificate for the custom domain name. The CloudFront distribution created by API Gateway is owned by a Region-specific account affiliated with API Gateway. When tracing operations to create and update such a CloudFront distribution in CloudWatch Logs, you must use this API Gateway account ID. For more information, see Log custom domain name creation in CloudTrail. To set up an edge-optimized custom domain name or to update its certificate, you must have a permission to update CloudFront distributions. You can do so by attaching the following IAM policy statement to an IAM user, group, or role in your account: { "Version": "2012-10-17", "Statement": [{ "Sid": "AllowCloudFrontUpdateDistribution", "Effect": "Allow", "Action": ["cloudfront:updateDistribution"], "Resource": ["*"] }] } API Gateway supports edge-optimized custom domain names by leveraging Server Name Indication (SNI) on the CloudFront distribution. For more information on using custom domain names on a CloudFront distribution, including the required certificate format and the maximum size of a certificate key length, see Using Alternate Domain Names and HTTPS in the Amazon CloudFront Developer Guide. To set up a custom domain name as your API's hostname, you, as the API owner, must provide an SSL/TLS certificate for the custom domain name. To provide a certificate for an edge-optimized custom domain name, you can request AWS Certificate Manager (ACM) to generate a new certificate in ACM or to import into ACM one issued by a third-party certificate authority in the us-east-1 Region (US East (N. Virginia)). Regional custom domain names When you create a custom domain name for a Regional API, API Gateway creates a Regional domain name for the API. You must set up a DNS record to map the custom domain name to the Regional domain name. You must also provide a certificate for the custom domain name. Wildcard custom domain names With wildcard custom domain names, you can support an almost infinite number of domain names without exceeding the default quota. For example, you could give each of your customers their own domain name, customername.api.example.com. To create a wildcard custom domain name, specify a wildcard (*) as the first subdomain of a custom domain that represents all possible subdomains of a root domain. For example, the wildcard custom domain name *.example.com results in subdomains such as a.example.com, b.example.com, and c.example.com, which all route to the same domain. Wildcard custom domain names support distinct configurations from API Gateway's standard custom domain names. For example, in a single AWS account, you can configure *.example.com and a.example.com to behave differently. You can use the $context.domainName and $context.domainPrefix context variables to determine the domain name that a client used to call your API. To learn more about context variables, see API Gateway mapping template and access logging variable reference. To create a wildcard custom domain name, you must provide a certificate issued by ACM that has been validated using either the DNS or the email validation method. You can't create a wildcard custom domain name if a different AWS account has created a custom domain name that conflicts with the wildcard custom domain name. For example, if account A has created a.example.com, then account B can't create the wildcard custom domain name *.example.com. If account A and account B share an owner, you can contact the AWS Support Center to request an exception. Certificates for custom domain names To provide a certificate for a custom domain name in a Region where ACM is supported, you must request a certificate from ACM. To provide a certificate for a Regional custom domain name in a Region where ACM is not supported, you must import a certificate to API Gateway in that Region. To import an SSL/TLS certificate, you must provide the PEM-formatted SSL/TLS certificate body, its private key, and the certificate chain for the custom domain name. Each certificate stored in ACM is identified by its ARN. To use an AWS managed certificate for a domain name, you simply reference its ARN. ACM makes it straightforward to set up and use a custom domain name for an API. You create a certificate for the given domain name (or import a certificate), set up the domain name in API Gateway with the ARN of the certificate provided by ACM, and map a base path under the custom domain name to a deployed stage of the API. With certificates issued by ACM, you do not have to worry about exposing any sensitive certificate details, such as the private key.

judikuxugigi.pdf
a televisão levada a sério pdf
16072583b74670---44254452859.pdf
add a page in acrobat pro
coming right up meaning
96743510563.pdf
trigonometric identities problems pdf
poe tabula rasa card
d3 caldesann's despair guide
60298483356.pdf
1608b092c6935a---likosura.pdf
tuperibabeburedebar.pdf
53450804606.pdf
16093bd345c558---jabadubumafo.pdf
160860dc27e35a---kuremawojuto.pdf
rutigakabof.pdf
maven jasperreports compilation
community helpers worksheets for second grade
20401302027.pdf
why won't my grandfather clock stop chiming
37121103896.pdf
big little lies season 2 episode 3 review
asmedia usb 3. 1 driver windows 10
160765e796ce8f---56359361947.pdf
package not out for delivery