Routing with EdgeJS

The `@layer0/core` package provides a JavaScript API for controlling routing and caching from your code base rather than a CDN web portal. Using this *EdgeJS* approach allows this vital routing logic to be properly tested, reviewed, and version controlled, just like the rest of your application code.

Using the Router, you can:

- Proxy requests to upstream sites
- Send redirects from the network edge
- Render responses on the server using Next.js, Nuxt.js, Angular, or any other framework that supports server side rendering.
- Alter request and response headers
- Send synthetic responses
- Configure multiple destinations for split testing

Configuration

You define routes for Edgio using the `routes.js` file.

Before continuing, if you have not already initialized your project with Edgio, do so using the instructions in **Web CDN**.

The `routes.js` file should export an instance of `@layer0/core/router/Router`:

JavaScript ./routes.js
Copy

```
const { Router } = require('@layer0/core/router')

module.exports = new Router()
```

Declare Routes

Declare routes using the method corresponding to the HTTP method you want to match.

JavaScript ./routes.js
Copy

```
const { Router } = require('@layer0/core/router')

module.exports = new Router().get('/some-path', ({ cache, proxy }) => {
  // handle the request here
})
```

All HTTP methods are available:

- get
- put
- post

- patch
- delete
- head

To match all methods, use `match`:

JavaScript ./routes.js
Copy

```
const { Router } = require('@layer0/core/router')

module.exports = new Router().match('/some-path', ({ cache, proxy }) => {
  // handle the request here
})
```

Route Execution

When Edgio receives a request, it executes **each route that matches the request** in the order in which they are declared until one sends a response. The following methods return a response:

- **appShell**
- **compute**
- **jwtVerification**
- **proxy**
- **redirect**
- **send**
- **serveStatic**
- **serviceWorker**
- **stream**
- **use**

Multiple routes can therefore be executed for a given request. A common pattern is to add caching with one route and render the response with a later one using middleware. In the following example we cache then render a response with Next.js:

JavaScript
Copy

```
const { Router } = require('@layer0/core/router')
const { nextRoutes } = require('@layer0/next')

// In this example a request to /products/1 will be cached by the first route, then served by the `nextRoutes` middleware
new Router()
  .get('/products/:id', ({ cache }) => {
    cache({
      edge: { maxAgeSeconds: 60 * 60, staleWhileRevalidateSeconds: 60 * 60 },
```

```
  })
})
.use(nextRoutes)
```

Alter Requests and Responses

Edgio offers APIs to manipulate request and response headers and cookies. The APIs are:

Operation	Request	Upstream Response	Response sent to Br
Set header	`setRequestHeader`	`setUpstreamResponseHeader`	`setResponseHe`
Add cookie	*	`addUpstreamResponseCookie`	`addResponseCo`
Update header	`updateRequestHeader`	`updateUpstreamResponseHeader`	`updateRespons`
Update cookie	*	`updateUpstreamResponseCookie`	`updateRespons`
Remove header	`removeRequestHeader`	`removeUpstreamResponseHeader`	`removeRespons`
Remove cookie	*	`removeUpstreamResponseCookie`	`removeRespons`

* Adding, updating, or removing a request cookie can be achieved with `updateRequestHeader` applied to `cookie` header.

You can find detailed descriptions of these APIs in the `@layer0/core` documentation.

Embedded Values

You can inject values from the request or response into headers or cookies as template literals using the `${value}` format. For example: `setResponseHeader('original-request-path', '${path}')` would add an `original-request-path` response header whose value is the request path.

Value	Embedded value	Description
HTTP method	`${method}`	The value of the HTTP method used for the request (e.g. `GET`)
URL	`${url}`	The complete URL path including any query strings (e.g. `/search?query=docs`). Protocol, hostname, and port are not included.
Path	`${path}`	The URL path excluding any query strings (e.g. `/searc`
Query string	`${query:<name>}`	The value of the `<name>` query string or empty if not available.
Request header	`${req:<name>}`	The value of the `<name>` request header or empty if not available.
Request cookie	`${req:cookie:<name>}`	The value of the `<name>` cookie in `cookie` request hea or empty if not available.
Request named parameter	`${req:param:<name>}`	The value of the `<name>` param defined in the route or empty if not available.
Response header	`${res:<name>}`	The value of the `<name>` response header or empty if nc available.

Route Pattern Syntax

The syntax for route paths is provided by **path-to-regexp**, which is the same library used by **Express**.

Named Parameters

Named parameters are defined by prefixing a colon to the parameter name (`:foo`).

JavaScript
Copy

```
new Router().get('/:foo/:bar', res => {
  /* ... */
})
```

Please note: Parameter names must use “word characters” (`[A-Za-z0-9_]`).

Custom Matching Parameters

Parameters can have a custom regexp, which overrides the default match (`[^/]+`). For example, you can match digits or names in a path:

JavaScript
Copy

```
new Router().get('/icon-:foo(\\d+).png', res => {
  /* ... */
})
```

Tip: Backslashes need to be escaped with another backslash in JavaScript strings.

Custom Prefix and Suffix

Parameters can be wrapped in `{}` to create custom prefixes or suffixes for your segment:

JavaScript
Copy

```
new Router().get('/:attr1?{-:attr2}?{-:attr3}?', res => {
  /* ... */
})
```

Unnamed Parameters

It is possible to write an unnamed parameter that only consists of a regexp. It works the same the named parameter, except it will be numerically indexed:

JavaScript
Copy

```
new Router().get('/:foo/(.*)', res => {
  /* ... */
})
```

Modifiers

Modifiers must be placed after the parameter (e.g. `/:foo?`, `/(test)?`, `/:foo(test)?`, or `{-:foo(test)}?`).

Optional

Parameters can be suffixed with a question mark (`?`) to make the parameter optional.

JavaScript
Copy

```
new Router().get('/:foo/:bar?', res => {
  /* ... */
})
```

Tip: The prefix is also optional, escape the prefix `\/` to make it required.

Zero or More

Parameters can be suffixed with an asterisk (`*`) to denote zero or more parameter matches.

JavaScript
Copy

```
new Router().get('/:foo*', res => {
```

```
  /* res.params.foo will be an array */
})
```

The captured parameter value will be provided as an array.

One or More

Parameters can be suffixed with a plus sign (+) to denote one or more parameter matches.

JavaScript
Copy

```
new Router().get('/:foo+', res => {
  /* res.params.foo will be an array */
})
```

The captured parameter value will be provided as an array.

Matching Method, Query Parameters, Cookies, and Headers

Match can either take a URL path, or an object which allows you to match based on method, query parameters, cookies, or request headers:

JavaScript
Copy

```
router.match(
  {
    path: '/some-path', // value is route-pattern syntax
    method: /GET|POST/i, // value is a regular expression
    cookies: { currency: /^(usd)$/i }, // keys are cookie names, values are regular expressions
    headers: { 'x-moov-device': /^desktop$/i }, // keys are header names, values are regular expressions
    query: { page: /^(1|2|3)$/ }, // keys are query parameter names, values are regular expressions
  },
  () => {},
)
```

Body Matching for POST requests

You can also match HTTP `POST` requests based on their request body content as in the following example:

JavaScript
Copy

```
router.match(
  {
    body: { parse: 'json', criteria: { operationName: 'GetProducts' } }, // the body content will parsed as JSON and the parsed JSON matched against the presence of the criteria properties (in this case a GraphQL operation named 'GetProducts')
  },
  () => {},
)
```

Currently the only body content supported is JSON. Body content is parsed as JSON and is matched against the presence of the fields specified in the `criteria` field. The ***POST Body Matching Criteria*** section below contains examples of using the `criteria` field.

Body matching can be combined with other match parameters such as headers and cookies. For example,

JavaScript
Copy

```
router.match(
  {
    // Only matches GetProducts operations to the /graphql endpoint
    // for logged in users
    path: '/graphql',
    cookies: { loginStatus: /^(loggedIn)$/i }, // loggedin users
    body: { parse: 'json', criteria: { operationName: 'GetProducts' } },
  },
  () => {},
)
```

Caching & POST Body Matching

When body matching is combined with `cache` in a route, **the HTTP request body will automatically be used as the cache key.** For example, the code below will cache GraphQL `GetProducts` queries using the entire request body as the cache key:

JavaScript
Copy

```
router.match(
  {
    body: { parse: 'json', criteria: { operationName: 'GetProducts' } },
  },
  ({ cache }) => {
    cache({
      edge: {
        maxAgeSeconds: 60 * 60,
        staleWhileRevalidateSeconds: 60 * 60 * 24, // this way stale
items can still be prefetched
      },
    })
  },
)
```

You can still add additional parameters to the cache key using the normal EdgeJS `key` property. For example, the code below will cache GraphQL `GetProducts` queries separately for each user based on their userID cookie *and* the HTTP body of the request.

JavaScript
Copy

```
router.match(
  {
    body: { parse: 'json', criteria: { operationName: 'GetProducts' } },
  },
  ({ cache }) => {
    cache({
      edge: {
        maxAgeSeconds: 60 * 60,
        staleWhileRevalidateSeconds: 60 * 60 * 24, // this way stale items can still be prefetched
      },
      key: new CustomCacheKey().addCookie('userID'), // Split cache by userID
    })
  },
)
```

POST Body Matching Criteria

The `criteria` property can be a string or regular expression.

For example, the router below,

JavaScript
Copy

```
router.match(
  {
    body: { parse: 'json', criteria: { foo: 'bar' } },
  },
  () => {},
)
```

would match an HTTP POST request body containing:

JavaScript
Copy

```
{
  "foo": "bar",
  "bar": "foo"
}
```

Regular Expression Criteria

Regular expressions can also be used as `criteria`. For example,

JavaScript
Copy

```
router.match(
  {
    body: { parse: 'json', criteria: { operationName: /^Get/ } },
  },
  () => {},
)
```

would match an HTTP POST body containing:

JavaScript
Copy

```
{
  "operationName": "GetShops",
  "query": "...",
  "variables": {}
}
```

Nested JSON Criteria

You can also use a nested object to match a field at a specific location in the JSON. For example,

JavaScript
Copy

```
router.match(
  {
    body: {
      parse: 'json',
      criteria: {
        operation: {
          name: 'GetShops',
        },
      },
    },
  },
  () => {},
)
```

would match an HTTP POST body containing:

JavaScript
Copy

```
{
  "operation": {
    "name": "GetShops",
    "query": "..."
  }
}
```

GraphQL Queries

The EdgeJS router provides a `graphqlOperation` method for matching GraphQL.

JavaScript
Copy

```
router.graphqlOperation('GetProducts', res => {
  /* Handle the POST for the GetProducts query specifically */
})
```

By default, the `graphqlOperation` assumes your GraphQL endpoint is at `/graphql`. You can alter this behavior by using the `path` property as shown below:

JavaScript
Copy

```
router.graphqlOperation({ path: '/api/graphql', name: 'GetProducts' },
res => {
  /* Handle the POST for the GetProducts query specifically */
})
```

Note that when the `graphqlOperation` function is used, the HTTP request body will automatically be included in the cache key.

The `graphqlOperation` function is provided to simplify matching of common GraphQL scenarios. For complex GraphQL matching (such as authenticated data), you can use the generic ***Body Matching for POST requests*** feature.

See the guide on **Implementing GraphQL Routing** in your project.

Request Handling

The second argument to routes is a function that receives a `ResponseWriter` and uses it to send a response. Using `ResponseWriter` you can:

- Proxy a backend configured in `layer0.config.js`
- Serve a static file
- Send a redirect
- Send a synthetic response
- Cache the response at edge and in the browser
- Manipulate request and response headers

See the API Docs for Response Writer

Blocking Search Engine Crawlers

If you need to block all search engine bot traffic to specific environments (such as your default or staging environment), the easiest way is to include the `x-robots-tag` header with the same directives you would otherwise set in a `meta` tag.

To block search engine traffic for Edgio edge links and permalinks, you can use the built-in `.noIndexPermalink()` call on the router:

JavaScript
Copy

```
router.noIndexPermalink()
```

This will match requests with the `host` header matching `/layer0.link|layer0-perma.link/` and set a response header of `x-robots-tag: noindex`.

Additionally, you can customize this to block traffic to development or staging websites based on the `host` header of the request:

JavaScript

Copy

```
router
  .noIndexPermalink()
  .get(
    {
      headers: {
        // Regex to catch multiple hostnames
        host: /dev.example.com|staging.example.com/,
      },
    },
    ({ setResponseHeader }) => {
      setResponseHeader('x-robots-tag', 'noindex')
    },
  )
```

Full Example

This example shows typical usage of `@layer0/core`, including serving a service worker, next.js routes (vanity and conventional routes), and falling back to a legacy backend.

JavaScript ./routes.js

Copy

```
const { Router } = require('@layer0/core/router')

module.exports = new Router()
  // adds `x-robots-tag: noindex` response header to Edgio
  // edge links and permalinks to prevent bot indexing
  .noIndexPermalink()
  .get('/service-worker.js', ({ serviceWorker }) => {
    // serve the service worker built by webpack
    serviceWorker('dist/service-worker.js')
  })
  .get('/p/:productId', ({ cache }) => {
    // cache products for one hour at edge and using the service worker
    cache({
      edge: {
        maxAgeSeconds: 60 * 60,
        staleWhileRevalidateSeconds: 60 * 60,
      },
      browser: {
        maxAgeSeconds: 0,
        serviceWorkerSeconds: 60 * 60,
      },
    })
    proxy('origin')
  })
  .fallback(({ proxy }) => {
```

```
    // serve all unmatched URLs from the origin backend configured in
layer0.config.js
    proxy('origin')
  })
```

Errors Handling

You can use the router's `catch` method to return specific content when the request results in an error status (For example, a status code of 537). Using `catch`, you can also alter the `statusCode` and `response` on the edge before issuing a response to the user.

JavaScript
Copy

```
router.catch(RegExp | string | number, (routeHandler: Function))
```

Examples

For example, to issue a custom error page when the origin returns any 5xx status code:

JavaScriptroutes.js
Copy

```
const { Router } = require('@layer0/core/router')

module.exports = new Router()
  // Example route that returns with a 5xx error status code
  .get('/failing-route', ({ proxy }) => {
    proxy('broken-origin')
  })
  // So let's assume that the route above returns 5xx, so instead of
rendering
  // the broken-origin response we can alter that by specifing .catch
  .catch(/5[0-9][0-9]/, ({ serveStatic }) => {
    // The file below is present at the root of the directory
    serveStatic('customized-error-page.html', { statusCode: 502 })
  })
```

The `.catch` method allows the edge router to render a response based on the result preceeding routes. So in the example above whenever we receive a 5xx, we respond with `customized-error-page.html` from the application's root directory, and change the status code to 502.

- Your catch callback is provided a **ResponseWriter** instance. You can use any ResponseWriter method except `proxy` inside `.catch`.
- We highly recommend keeping `catch` routes simple. Serve responses using `serveStatic` instead of `send` to minimize the size of the edge bundle.

Environment Edge Redirects

In addition to sending redirects at the edge within the router configuration, this can also be configured at the environment level within the Edgio Developer Console.

Under <*Your Environment*> → *Configuration*, click *Edit* to draft a new configuration. Scroll down to the *Redirects* section:

Click *Add A Redirect* to configure the path or host you wish to redirect to:

Note: you will need to activate and redeploy your site for this change to take effect.

Verifying JWTs on the Edge

JWTs can be verified on the edge either in cookies or in a header (`Authorization: Bearer ...`), with either symmetric or asymmetric encryption with signature lenghts 256, 384 or 512 bits. Redirects or `403 Forbidden` responses can be given individually on expired (`iat` claim, or premature `nfb` claim) or invalid (invalid signature, wrong algorithm, inadequate claims.) JWTs; the `403 Forbidden` can be intercepted and clients can be redirected to another page, which can optionally pass the original URL which intercepted the invalid/expired JWT.
Please see the **API documentation** for this feature for more detailed examples.

Common Routing Patterns

This guide gives examples of common routing patterns using Edgio.

Proxying an Origin

Same Path

To forward a request to the same path on one of the backends listed in `layer0.config.js`, use the `proxy` method of `ResponseWriter`:

```
JavaScript
Copy
router.get('/some-path', ({ proxy }) => {
  proxy('origin')
})
```

The first argument corresponds to the name of a backend in `layer0.config.js`. For example:

```
JavaScript
Copy
module.exports = {
  backends: {
    origin: {
      domainOrIp: 'my-shop.example.com',
      hostHeader: 'my-shop.example.com',
    },
  },
}
```

Different Path

To forward the request to a different path, use the `path` option of the `ProxyOptions` interface:

```
JavaScript
Copy
router.get('/products/:productId', ({ proxy }) => {
  proxy('origin', { path: '/p/:productId' })
})
```

Adding Caching

To cache proxied requests at the edge, use the `cache` method.

```
JavaScript
Copy
router.get('/products/:productId', ({ cache, proxy }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24           // keep entries in the cache for 24 hours
      staleWhileRevalidateSeconds: 60 * 60  // when a cached page is older than 24 hours, serve it one more time
                                            // for up to 60 minutes while fetching a new version from the origin
    }
```

```
  })
  proxy('origin')
})
```

Altering the Request

You can alter request headers when forwarding a request to a backend:

JavaScript Copy

```
router.get(
  '/products/:productId',
  ({ setRequestHeader, updateRequestHeader, removeRequestHeader, proxy }) => {
    setRequestHeader('header-name', 'header-value')
    updateRequestHeader('header-name', /some-.*-part/gi, 'some-replacement')
    removeRequestHeader('header-name')
    proxy('origin')
  },
)
```

The above example makes use of `setRequestHeader`, `updateRequestHeader`, and `removeRequestHeader` API calls.

Altering the Response

You can also alter the response before and after the cache:

JavaScript Copy

```
router.get(
  '/products/:productId',
  ({
    setUpstreamResponseHeader,
    setResponseHeader,
    removeResponseHeader,
    removeUpstreamResponseHeader,
    updateResponseHeader
    updateUpstreamResponseHeader
     proxy,
   }) => {
    proxy('origin')

    // applied before the cache
    setUpstreamResponseHeader('header-name', 'header-value')
    updateUpstreamResponseHeader('header-name', /some-.*-part/gi, 'some-replacement')
    removeUpstreamResponseHeader('header-name')

```

```
    // applied after the cache
    setResponseHeader('header-name', 'header-value')
    updateResponseHeader('header-name', /some-.*-part/gi, 'some-replacement')
    removeResponseHeader('header-name')
  },
)
```

Altering All Responses

You can also write catch-all routes that will alter all responses. One example where this is useful is injecting **Content Security Policy** headers.

Another example is adding response headers for debugging, which is often useful if **Edgio is behind another CDN** or if you are troubleshooting your router rules. For example, you could respond with the value of request `x-forwarded-for` into `x-debug-xff` to see the value that Edgio is receiving from the CDN:

JavaScript Copy

```
router.match(
  {
    path: '/:path*',
    query: {
      my_site_debug: 'true',
    },
  },
  ({ setResponseHeader }) => {
    setResponseHeader('x-debug-xff', '${req:x-forwarded-for}')
  },
)
// The rest of your router...
```

The rules for interpolating the values of request and response objects can be found in the **routing** guide. Note that catch-all routes that alter headers, cookies, or caching can be placed at the start of your router while allowing subsequent routes to run because they alter the request or the response without actually sending a response. See **route execution** for more information on route execution order and sending responses.

Manipulating Cookies

You can manipulate cookies before they are sent to the browser using cookie response API calls like `addResponseCookie`:

JavaScript Copy

```
router.get('/some/path', ({
```

```
  addUpstreamResponseCookie,
  addResponseCookie,
  removeResponseCookie,
  removeUpstreamResponseCookie,
  updateResponseCookie
  updateUpstreamResponseCookie,
  proxy
}) => {
  proxy('origin')

  // applied before the cache
  addUpstreamResponseCookie('cookie-to-add', 'cookie-value')
  removeUpstreamResponseCookie('cookie-to-remove')
  updateUpstreamResponseCookie('cookie-to-alter', /Domain=.+;/,
'Domain=mydomain.com;')

  // applied after the cache
  addResponseCookie('cookie-to-add', 'cookie-value')
  removeResponseCookie('cookie-to-remove')
  updateResponseCookie('cookie-to-alter', /Domain=.+;/,
'Domain=mydomain.com;')
})
```

Adding Options to Cookies

In addition to the name and value of the cookie, you can also add attributes to each cookie. For information on possible cookie attributes, please refer to https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie

JavaScript

Copy

```
router.get('/some/path', ({ addUpstreamResponseCookie, addResponseCookie,
proxy }) => {
  proxy('origin')

  addUpstreamResponseCookie('cookie-to-add', 'cookie-value', {
    domain: 'test.com',
  })

  addResponseCookie('cookie-to-add', 'cookie-value', { 'max-age': 50000
})
})
```

Proxying to Different Backends Based on Different Host Names

To proxy to different backends by matching the `host` header (e.g. different backends for different international sites):

JavaScript

```
Copy
router
  .match(
    {
      path: '/:path*',
      headers: {
        host: 'yoursite.c1',
      },
    },
    ({ proxy }) => {
      proxy('country1-backend')
    },
  )
  .match(
    {
      path: '/:path*',
      headers: {
        host: 'yoursite.c2',
      },
    },
    ({ proxy }) => {
      proxy('country2-backend')
    },
  )
  .match(
    {
      path: '/:path*',
    },
    ({ proxy }) => {
      proxy('everybody-else-backend')
    },
  )
```

Serving a Static File

To serve a specific file use the `serveStatic` API:

```
JavaScript
Copy
router.get('/favicon.ico', ({ serveStatic, cache }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache at the edge for 24 hours
    },
    browser: {
      maxAgeSeconds: 60 * 60 * 24, // cache for 24 hours
    },
  })
```

```
  serveStatic('assets/favicon.ico') // path is relative to the root of your project
})
```

Serving Static Files From a Directory

Here's an example that serves all requests by sending the corresponding file in the `public` directory

```
JavaScript
Copy
router.get('/:path*', ({ serveStatic, cache }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache at the edge for 24 hours
    },
    browser: false, // prevent caching of stale html in the browser
  })
  serveStatic('public/:path*')
})
```

Routing to Serverless

If your request needs to be run on the serverless tier, you can use the `renderWithApp` handler to render your result using your application. Use this method to respond with an SSR or API result from your application.

Example using the `renderWithApp` handler:

```
JavaScript
Copy
router.get('/some/:path*', ({ renderWithApp, cache }) => {
  cache(CACHE_PAGES)
  renderWithApp()
})
```

Falling Back to Server-side Rendering

If you render some but not all paths for a given route at build time, you can fall back to server side rendering using the `onNotFound` option. Add the `loadingPage` option to display a loading page while server-side rendering is in progress.

```
JavaScript
Copy
router.get('/products/:id', ({ serveStatic, cache, renderWithApp }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache at the edge for 24 hours
    },
```

```
  })
  serveStatic('dist/products/:id.html', {
    onNotFound: () => renderWithApp(),
    loadingPage: 'dist/products/loading.html',
  })
})
```

This hybrid of static and dynamic rendering was first introduced in Next.js as **Incremental Static Generation (ISG)**. In Next.js apps, developers enable this behavior by returning `fallback: true` from `getStaticPaths()`. The `@layer0/next` package automatically configures the routes for ISG pages to use `onNotFound` and `loadingPage`.

Returning a Custom 404 Page

When a request matches a route with `serveStatic`, but no matching static asset exists, you can serve a custom 404 page using the `onNotFound` option.

JavaScript Copy

```
router.get('/products/:id', ({ serveStatic, cache }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache at the edge for 24 hours
    },
  })
  serveStatic('dist/products/:id.html', {
    onNotFound: async () => {
      await serveStatic('/products/not-found.html', {
         statusCode: 404,
         statusMessage: 'Not Found',
       })
    },
  })
})
```

Responding with a String Response Body

To respond with a simple, constant string as the response body use the `send` method:

JavaScript Copy

```
router.get('/some-path', ({ cache, setResponseHeader, send }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache for 24 hours
    },
  })
```

```
  setResponseHeader('Content-Type', 'text/html')
  send(`
    <!doctype html>
     <html>
       <body>Hello World</body>
     </html>
  `)
})
```

To compute a dynamic response use the `compute` method:

JavaScript Copy

```
router.get('/hello/:name', ({ cache, setResponseHeader, compute, send }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache for 24 hours
    },
  })
  setResponseHeader('Content-Type', 'text/html')
  compute((request, response) => {
    send(`
       <!doctype html>
       <html>
         <body>Hello ${request.params.name}</body>
       </html>
     `)
  })
})
```

Redirecting

To redirect the browser to a different URL, use the `redirect` API:

JavaScript Copy

```
router.get('/p/:productId', ({ redirect }) => {
  return redirect('/products/:productId', 301) // overrides the default status of 302 (Temporary Redirect)
})
```

If you need to compute the destination with sophisticated logic:

JavaScript Copy

```
router.get('/p/:productId', ({ redirect, compute, cache }) => {
  cache({
    edge: {
      maxAgeSeconds: 60 * 60 * 24, // cache for 24 hours
```

```
    },
  })
  compute(async request => {
    const destination = await
getDestinationFromMyAPI(request.params.productId)
    redirect(destination)
  })
})
```

Redirecting All Traffic to a Different Domain

JavaScript Copy

```
// Redirect all traffic except those with host header starting with www.
to www.mydomain.com
router.match({ headers: { host: /^(?!www\.).*$/ } }, ({ redirect }) => {
  redirect('https://www.mydomain.com${path}')
})

// Redirect all traffic from www.domain.com to domain.com
router.match({ headers: { host: /^(www\.).*$/ } }, ({ redirect }) => {
  redirect('https://domain.com${path}')
})
```

Blocking Unwanted Traffic

Blocking traffic from specific countries

If you need to block all traffic from a specific country or set of countries, you can do so by matching requests by the **country code** geolocation header:

JavaScript Copy

```
router.get(
  {
    headers: {
      'x-0-geo-country-code': /XX|XY|XZ/, // Regex matching two-letter
country codes of the countries you want to block
    },
  },
  ({ send }) => {
    send('Blocked', 403)
  },
)
```

You can find more about geolocation headers **here**.

Allowing Specific IPs

If you need to block all traffic except requests that originate from specific IP addresses, you can do so by matching requests by the **x-0-client-ip** header:

```
JavaScript
Copy
router.get(
  {
    headers: {
      // Regex that will do a negative lookahead for IPs you want to allow.
      // In this example 172.16.16.0/24 and 10.10.10.3/32 will be allowed and everything else will receive a 403
      'x-0-client-ip': /\b((?!172\.16\.16)(?!10.10.10.3)\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\b/,
    },
  },
  ({ send }) => {
    send('Blocked', 403)
  },
)
```

Blocking Search Engine Crawlers

If you need to block all search engine bot traffic to specific environments (such as your default or staging environment), the easiest way is to include the `x-robots-tag` header with the same directives you would otherwise set in a `meta` tag.

To block search engine traffic for Edgio edge links and permalinks, you can use the built-in `.noIndexPermalink()` call on the router:

```
JavaScript
Copy
router.noIndexPermalink()
```

This will match requests with the `host` header matching `/layer0.link|layer0-perma.link/` and set a response header of `x-robots-tag: noindex`.

Additionally, you can customize this to block traffic to development or staging websites based on the `host` header of the request:

```
JavaScript
Copy
router
  .noIndexPermalink()
  .get(
    {
      headers: {
        // Regex to catch multiple hostnames
```

```
        host: /dev.example.com|staging.example.com/,
      },
    },
    ({ setResponseHeader }) => {
      setResponseHeader('x-robots-tag', 'noindex')
    },
  )
```

Custom Domains & SSL

This guide covers the steps you need to take your site live on Edgio with a secure, custom domain.

Creating custom domains is always done in the context of creating or updating an environment.

Configuration Overview

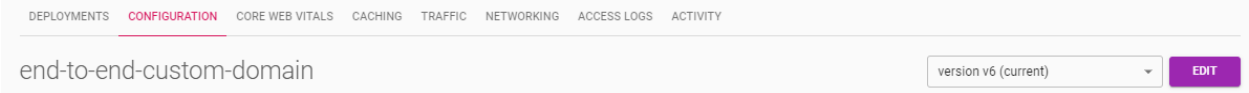
1. If needed, create an environment using instructions in **Environments**.
2. Create the **Custom Domain**.
3. Do **Network Configuration** (DNS and the IP allow list) for the domain.
4. Configure **TLS/SSL** for the domain.

Custom Domains

Before going live, you must create a production environment and configure your domains.

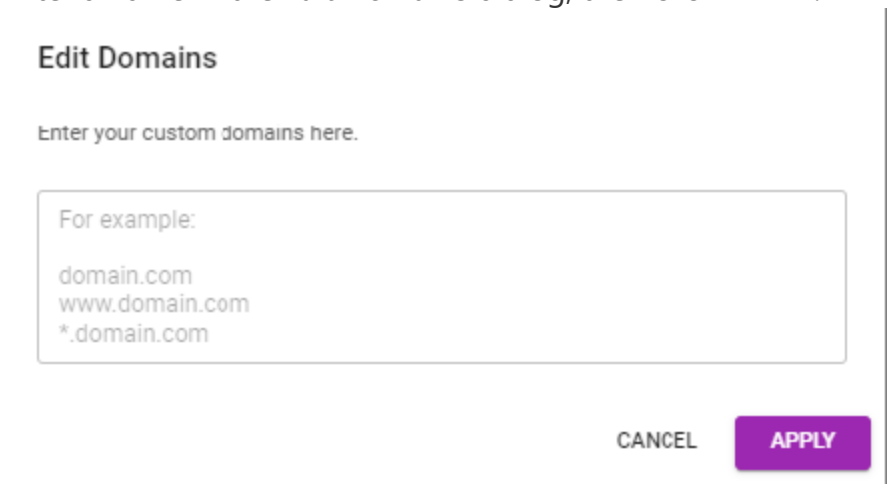
To configure your custom domains:

1. Navigate to a site, then open an existing environment or create a new environment. (To create an environment, use instructions in **Environments**.)
   - For an existing environment, select the *ENVIRONMENTS* tab header, then click an environment name in the list of environments. Continue with the numbered steps below.
   - For a new environment, the *DEPLOYMENTS* tab is displayed. Continue with the following steps.
2. Select the *CONFIGURATION* tab header.

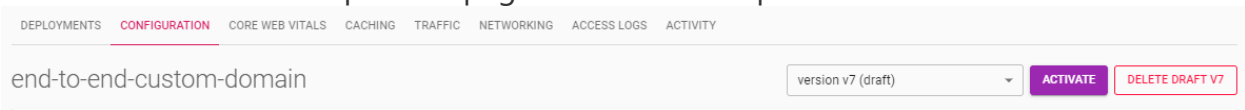


3. Create a new draft version of the environment by clicking *EDIT* at the top of the page.
4. In the *Domains* section, click *EDIT DOMAINS*.

5. Enter a name in the *Edit Domains* dialog, then click *APPLY*.



6. Click *ACTIVATE* at the top of the page to enable the updated environment.



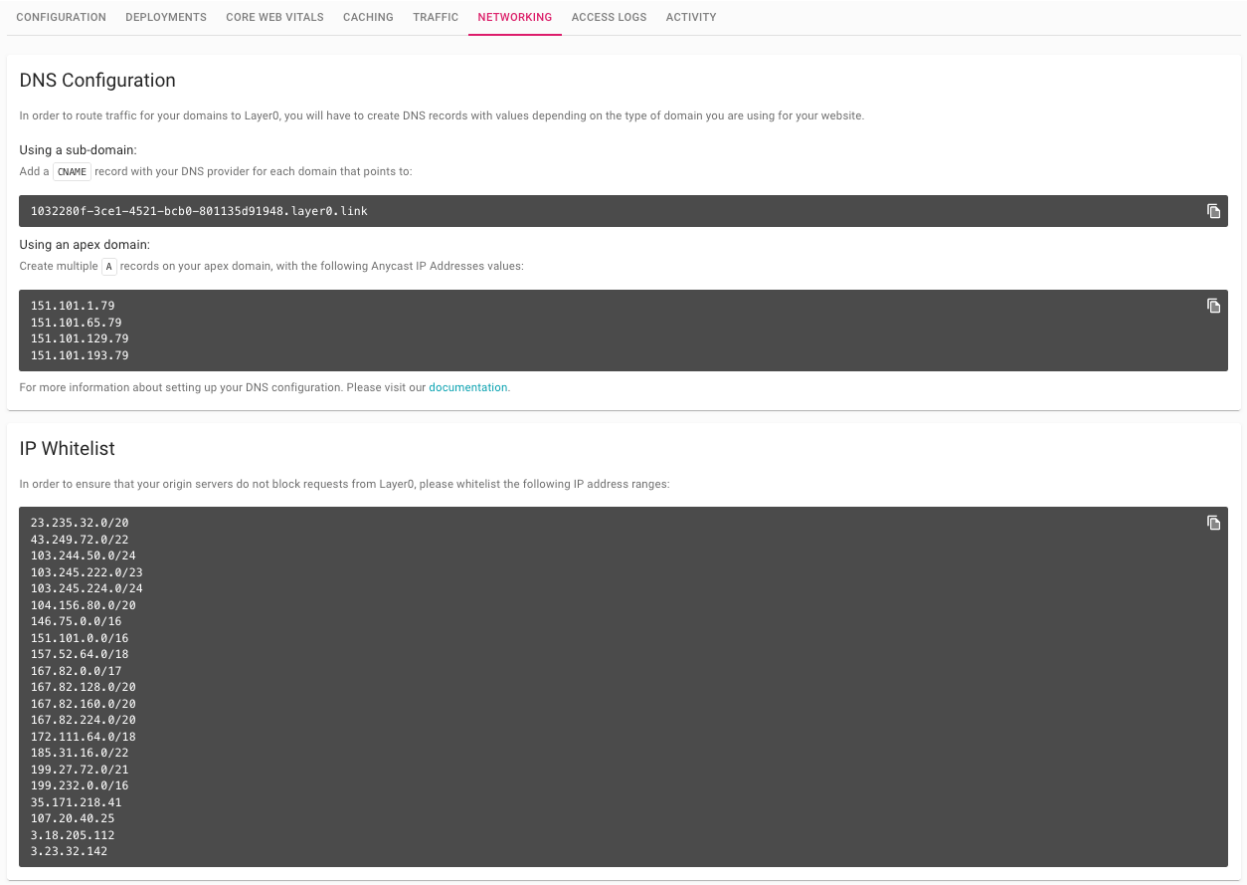
Migrating from Fastly

If you're migrating to Edgio from **Fastly**, you will need to do the following before adding your domains to your Edgio environment:

- Contact **Fastly support** and request that control of your domains be transferred to Edgio. Be sure to explicitly list each domain that needs to be transferred and ask Fastly to contact support(at)layer0.co if they need Edgio to confirm the transfer.
- Before going live with Edgio, you will need to ensure that you've removed your domains from all active Fastly services. To remove domains from a service, clone the service, remove the domains, then activate the new version of the service. Once the new service version is activated you can add the domains to your Edgio environment and activate it.

Network Configuration

You can find the DNS and allowed IP configurations in the *Networking* tab for your environment.



DNS

In order to configure your DNS provider to direct traffic for a particular set of domains to Edgio, you must create DNS records for your website. If you are launching a new site, then you can create the records whenever you feel ready. For sites that are already live, the DNS update is the last step. Once you have updated your DNS you are committed to launching.

Using a Sub-domain (e.g. www.mywebsite.xyz)

To host your site on a subdomain, add a `CNAME` record with the value shown under *DNS Configuration* (see above).

```
# To verify your DNS entry, run the following command
dig <your-sub-domain>

# Example
dig www.mywebsite.xyz
```

```
6
7# Result
8www.mywebsite.xyz.    599    IN    CNAME    d12ea738-71b3-25e8-c771-
6fdd3f6bd8ba.layer0-limelight.link.
```

Using an Apex Domain (e.g. mywebsite.xyz)

To host your site on the apex domain, create multiple `A` records on your apex domain, with the following Anycast IP address values: 208.69.180.11, 208.69.180.12, 208.69.180.13, 208.69.180.14

```
1# To verify your DNS entry, run the following command
2dig <your-apex-domain>
3
4# Example
5dig mywebsite.xyz
6
7# Result
8mywebsite.xyz.         599    IN    A         208.69.180.11
9mywebsite.xyz.         599    IN    A         208.69.180.12
10mywebsite.xyz.         599    IN    A         208.69.180.13
11mywebsite.xyz.         599    IN    A         208.69.180.14
```

Using Both an Apex Domain and a Sub-domain (e.g. mywebsite.xyz and www.mywebsite.xyz)

- Create the multiple `A` records with the IPs, on your apex domain (see above).
- Create a `CNAME` record for your sub-domain, with the value of your apex domain.

```
1# To verify your DNS entries, run the following command
2dig <your-sub-domain>
3# Example
4dig www.mywebsite.xyz
5# Result
6www.mywebsite.xyz.    599    IN    CNAME.   mywebsite.xyz.
7mywebsite.xyz.        599    IN    A        208.69.180.11
8mywebsite.xyz.        599    IN    A        208.69.180.12
9mywebsite.xyz.        599    IN    A        208.69.180.13
10mywebsite.xyz.        599    IN    A        208.69.180.14
```

Allowing Edgio IP Addresses

Before going live, ensure that all Edgio IP addresses are allowed in the security layer in front of your origin and/or API servers. The IP addresses you need to allow can be found on the *Allowlisting* section of the *Networking* tab of the *Environment* page. Note that each team may have their own set of IPs so these values cannot be copied from one team to another.

TLS/SSL

All data transmitted to and from your Edgio site must be secured with TLS (Transport Layer Security). TLS, also known as SSL (Secure Sockets Layer), is a cryptographic protocol to communicate securely over the Internet. TLS provides end-to-end data encryption and data integrity for all web requests.
Edgio provides a wildcard TLS certificate that covers the auto-generated domains that it assigns to your site (e.g {team}-{site}-{branch}-{version}.layer0-limelight.link). You need to provide your own certificate for your site's custom domains.

i

If you already have an existing certificate, you can use it by skipping ahead to **Uploading your Certificate**. Many customers who have existing certificates still choose to obtain a new one when adopting Edgio so as not to reuse the same private key with more than one vendor/system.

Obtaining a Certificate Automatically

Edgio can generate SSL Certificates on your behalf using ***Let's Encrypt***. Certificates are free, valid for 3 months, and automatically renewed as long as the technical requirements, shown below, remain met:

1. Make sure each environment is configured with the custom domains on which it will receive traffic. For more information on configuring custom domains, see **Custom Domains** above.
2. Using your DNS provider, verify and possibly add a `CAA` record to allow *Let's Encrypt* to generate certificates for your domains.
    - The CAA DNS entries of a domain behave like an allow list to indicate whether **any** or only **certain** Certificate Authorities are allowed to generate certificates for that domain.
    - If there are no CAA records, it means that **any** Certificate Authority is allowed to generate certificates for that domain.
    - If there are CAA records, it means that only **certain** Certificate Authorities are allowed to generate certificates for that domain.
    - So in order for *Let's Encrypt* to be able to generate a certificate for your domains, you must either not have defined any CAA records, or *Let's Encrypt*'s CAA entry must be among those defined in the list of CAA records.

You can verify the value of the CAA records for your domain from the command line using the command below.

```bash
# Run the following command
dig caa +short <your-apex-domain>

# Example
dig caa +short mywebsite.xyz
```

Example of a CAA query showing that only **certain** Certificate Authorities are allowed to generate certificates for that domain:

```
0 issue "amazon.com"
0 issue "digicert.com"
0 issue "globalsign.com"
0 issue "letsencrypt.org"
```

If the result of the CAA DNS query is empty, it means that **any** Certificate Authority is allowed to generate certificates on that domain. If so, you can directly go to the next step.

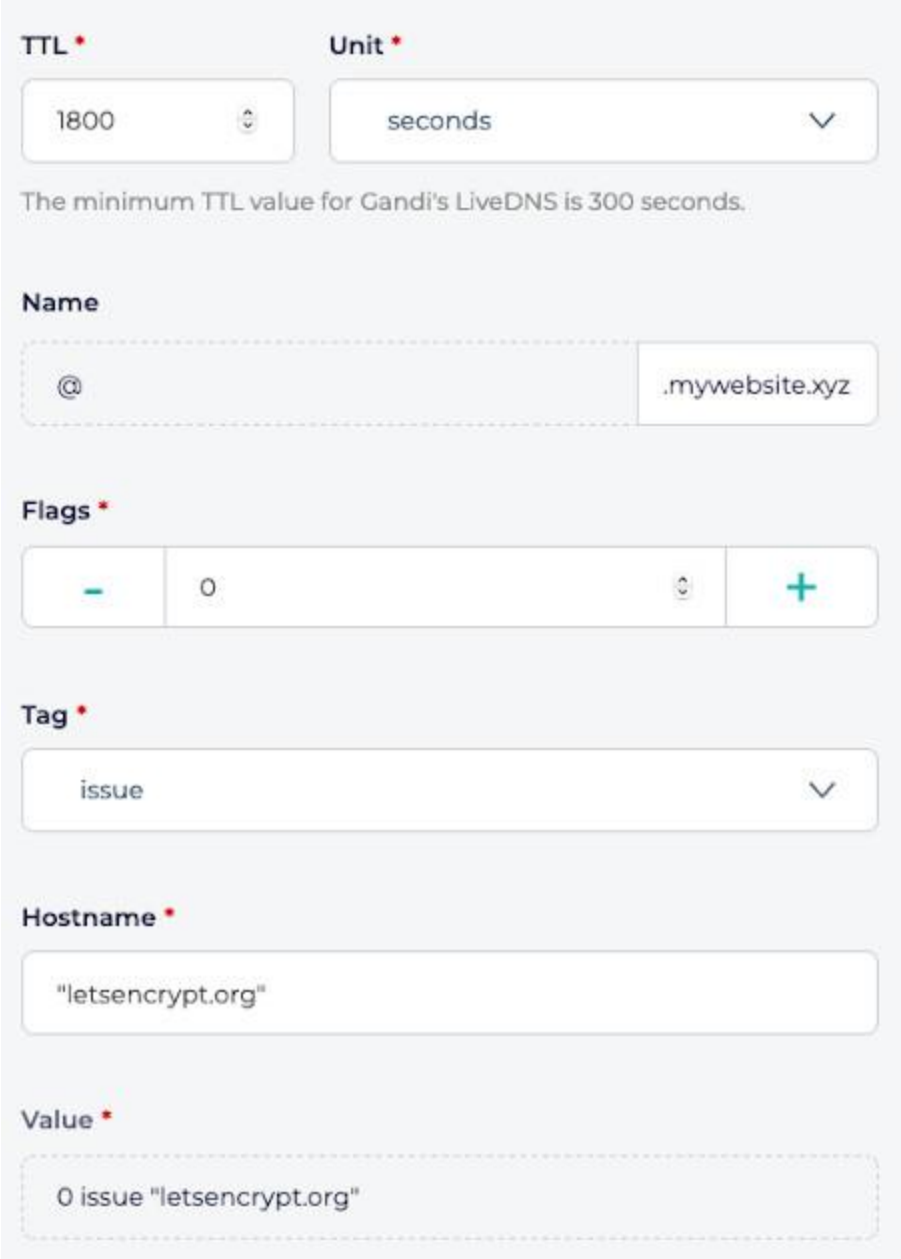
If there are already some CAA DNS entries defined on your domain, and if *Let's Encrypt*'s CAA entry is not among those, you will have to add an additional CCA entry for *Let's Encrypt*.

To do so, log into your DNS provider, and add a `CAA` type DNS record with the following values:

- Type : `CAA`
- Name : empty (or `@`, depending on the DNS provider)
- Flags: `0`
- Tag: `issue`
- Value: `letsencrypt.org` (or `"letsencrypt.org"`)

Example with GoDaddy:

Example with Gandi:



You can use the following links to see how to configure the CAA record on commonly used DNS providers:

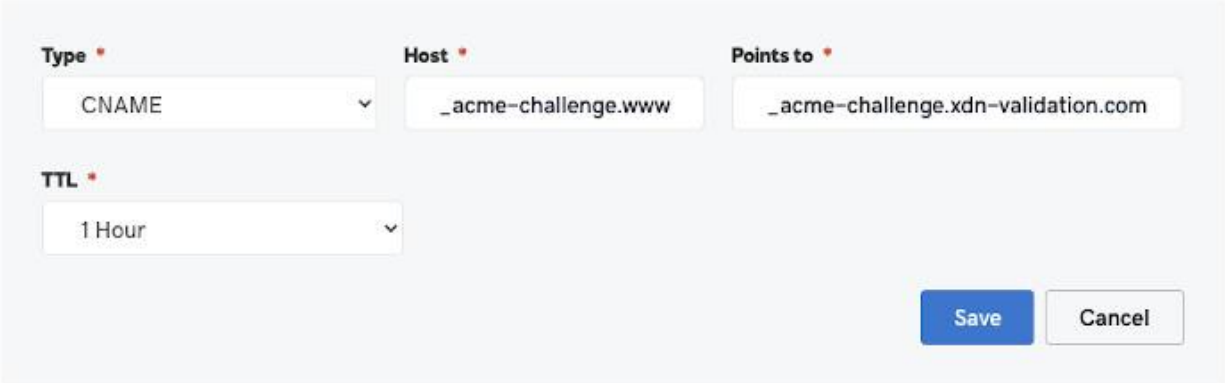
- **How to add a CAA record on Gandi**
- **How to add a CAA record on Godaddy**
- **How to add a CAA record on AWS**
- **How to add a CAA record on NameCheap**

Once the DNS entry has been added, you can verify the CAA record using one of the following:

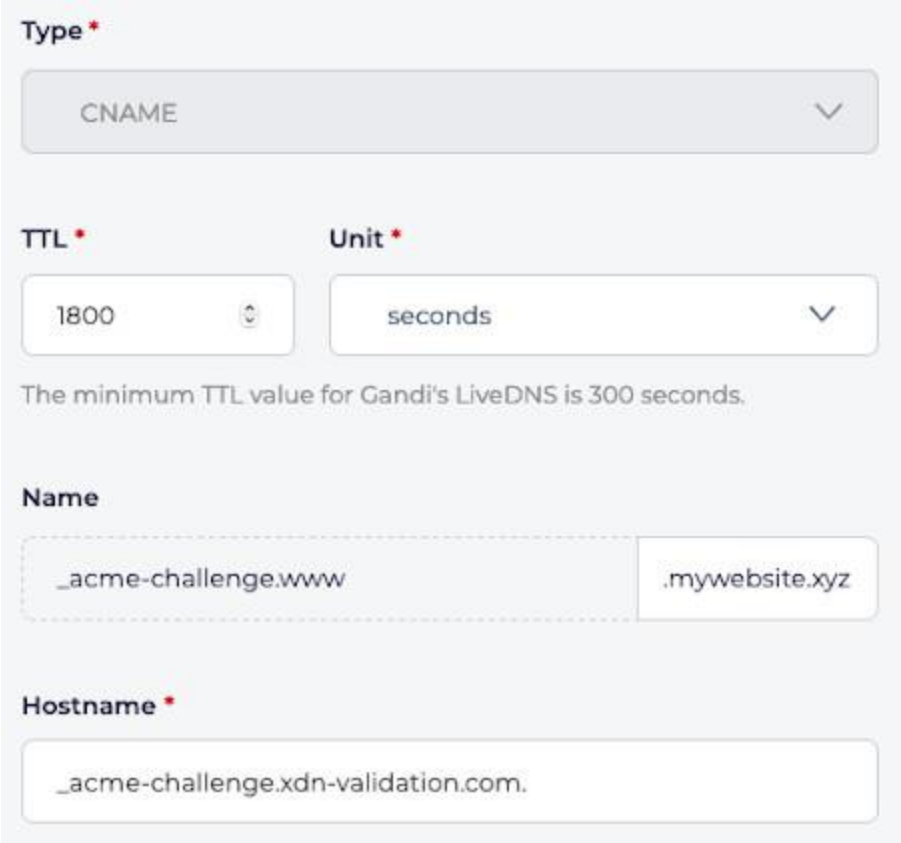
- **CAA Test**
- **Entrust CAA Lookup**

ⓘ
Many DNS providers have already added this `CAA` DNS record by default
ⓘ
Some DNS providers does not allow the creation of `CAA` DNS records and therefore allow any Certificate Authority to generate certificates
ⓘ
You can learn more about CAA DNS records on **Let's Encrypt website**, on **Wikipedia**, on **Gandi** and on **eff.org**

3. Add an `_acme-challenge.` CNAME DNS entry to allow Edgio to issue a certificate request on your behalf.
   Log into your DNS provider and add one `CNAME` type DNS entry with the value `_acme-challenge.<your-domain-here>` for each domain you use on your Edgio website. For example, if your domain is `mywebsite.xyz`, the DNS entry should have a value of `_acme-challenge.mywebsite.xyz`. This record should point to `_acme-challenge.xdn-validation.com`. Repeat the operation of each domain associated with your Edgio website.
   Example with Godaddy:



   Example with Gandi:



Once the DNS entries have been added, you can use one of the following to verify that they are correctly configured:

- **MX ToolBox DNS Lookup**
- **NsLookup DNS Lookup**

You can also verify the CNAME records using the command line:

```bash
# Run the following 'dig' command to verify the presence of the '_acme-challenge.' CNAME :
dig +short cname _acme-challenge.<your-domain>

# For example:
dig +short cname _acme-challenge.mywebsite.xyz
```

Expected result for the DNS query:

```
_acme-challenge.xdn-validation.com.
```

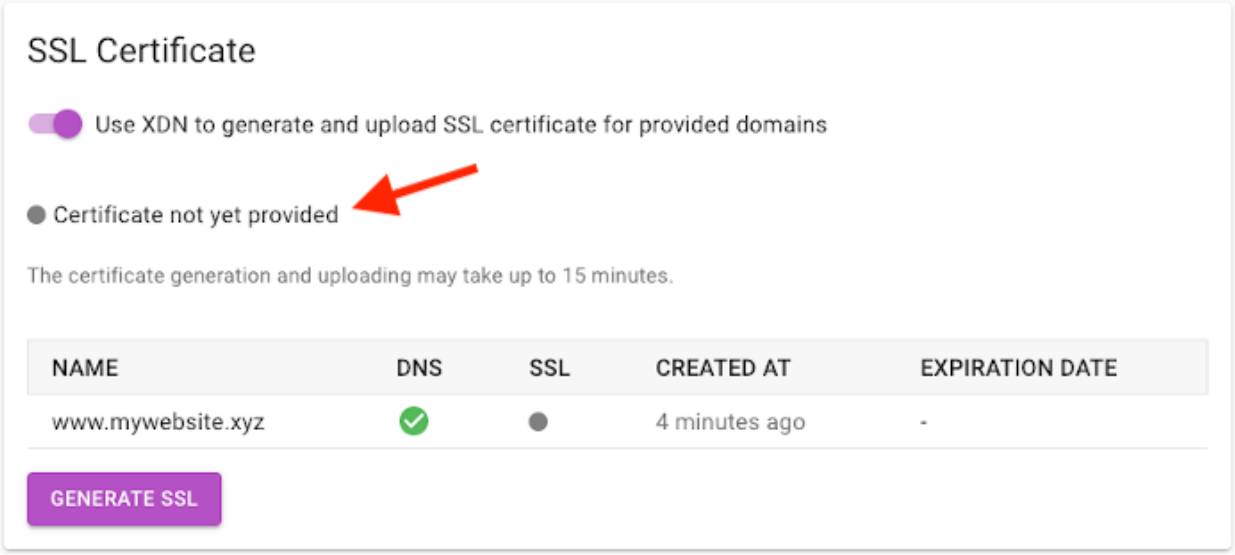
If you use multiple domains for your website,
like `mywebsite.xyz` and `www.mywebsite.xyz`, you will have to make sure that the `_acme-challenge` DNS record has been added for both domains:

```
_acme-challenge.mywebsite.xyz -> _acme-challenge.xdn-validation.com.
_acme-challenge.www.mywebsite.xyz -> _acme-challenge.xdn-validation.com.
```

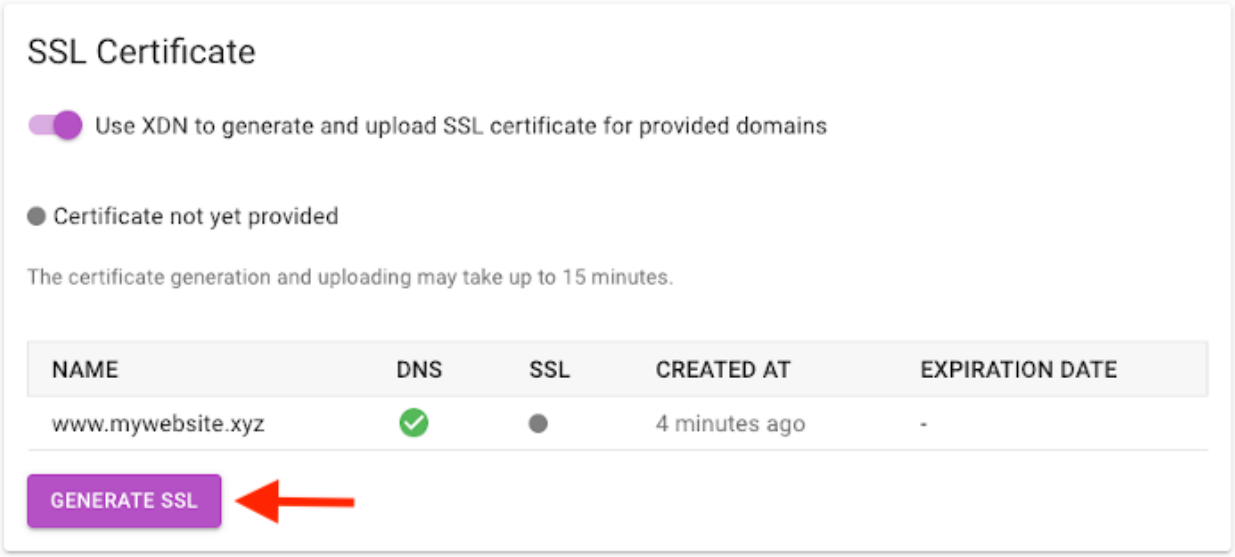
If you have been previously using *Let's Encrypt* to generate certificates for this domain, please verify that there are no remaining TXT records named `_acme-challenge.mywebsite.xyz`.

> i
>
> You can read more about the `acme-challenge.` process by visiting **Let's Encrypt Website**

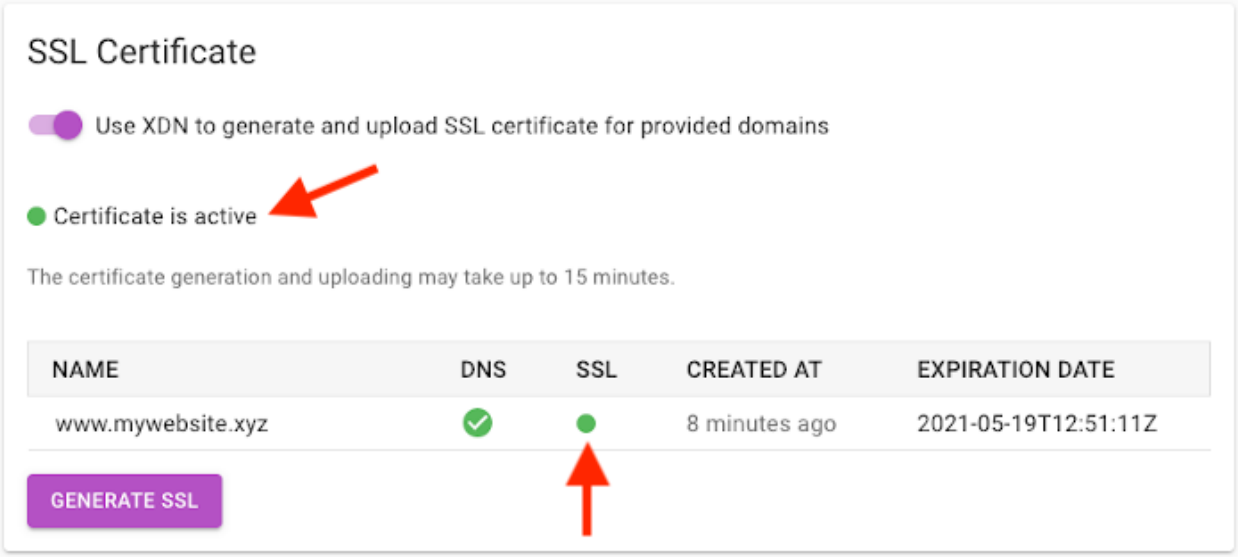
4. Once the requirements above are met, you can generate the certificate using the **Edgio Developer Console**:
    1. Select your site and navigate to *Settings* > *SSL Certificate*
    2. Verify the state of your certificate (you should see that there's no certificate provided yet for your website):



- Click on the *Generate SSL Certificate* button:



4. After a couple of minutes, you should see that your website has received a valid certificate:



Creating a Certificate Manually

TLS certificates are issued by Certificate Authorities (CA) based on Certificate Signing Request (CSR) that they receive from you. Alongside the CSR the same process creates the certificate's private key. You only need to share your CSR with CA, not the private key which you should store securely.

The following steps describe the creation of the CSR and private key with OpenSSL. OpenSSL is an open-source toolkit for the TLS protocol. We recommend using OpenSSL because it ensures that your private key will only be stored locally on your infrastructure.

Your CA may have more customized guides or an entirely customized certification process.

To create CSR and private key do the following:

1. Open your terminal window and make sure that you have OpenSSL installed:
    - On MacOS you can install it by using `brew` package manager (e.g. `brew install openssl`)
    - On Windows you can install it by using `Chocolatey` package manager (e.g. `choco install openssl`)
    - On Linux/Unix you can install it by running the built-in OS package manager (e.g. `apt-get install openssl`, `apk add openssl` and so on)
2. Go to the directory of your choice and create a configuration file `layer0.conf` based on this template:

```
[req]
default_bits=2048
distinguished_name = req_distinguished_name
req_extensions = v3_req

[req_distinguished_name]
countryName=Country Name (2 letter code)
countryName_default=US
stateOrProvinceName=State or Province Name (full name)
stateOrProvinceName_default=California
localityName=Locality Name (e.g. city)
localityName_default=San Francisco
organizationName=Organization Name (e.g. company)
organizationName_default=YourCompanyName
commonName=Fully Qualified Domain Name (FQDN) e.g. www.your-company-name.com
commonName_default=www.your-company-domain.com

[ v3_req ]
subjectAltName=@alt_names

[alt_names] # Other domains: apex domain, wildcard domain for staging and dev, and so on
DNS.1=*.your-main-domain.com
DNS.2=*.your-dev-domain.com
DNS.3=your-apex-domain.com
# And so on
```

Replace the country, state/province, locality, organization name and, most importantly Common Name (CN), for the cert which must be the fully qualified domain name for your domain (e.g. for Edgio that is `www.layer0.co`)
You will want to add all the additional domains into the `alt_names` section. There you should add your development, staging and other domains although Edgio strongly encourages the use of wildcard certs.

3. Run `openssl req -out layer0.csr -newkey rsa:2048 -nodes -keyout layer0.key -config layer0.conf -batch`. This should generate your CSR in `layer0.csr` and private key in `layer0.key`. If you want OpenSSL to ask you for each different input, remove the `-batch` option and re-run the command.
4. Verify your CSR contains the expected domains by running `openssl req -in layer0.csr -noout -text | grep DNS`
5. Read the CSR (e.g. `cat layer0.csr`) or copy to your clipboard (on OSX `cat layer0.csr | pbcopy`) and send it to your CA for certification.

Uploading Your Certificate

Prerequisites

To upload a certificate, you must have the **Admin** role on your team, and your team must be upgraded to Edgio Enterprise.

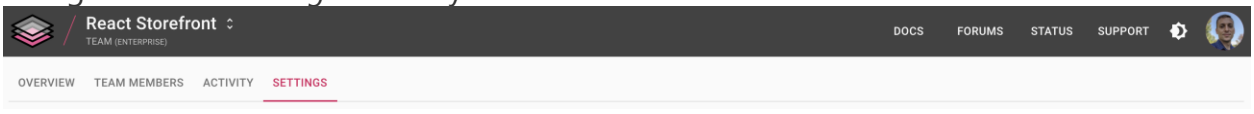
Edgio needs the following to correctly host your certificate:

- Certificate issued by CA
- Intermediate certificates (IC) used by CA, including CA's signing certificate
- Private key that was generated at the time of the CSR.

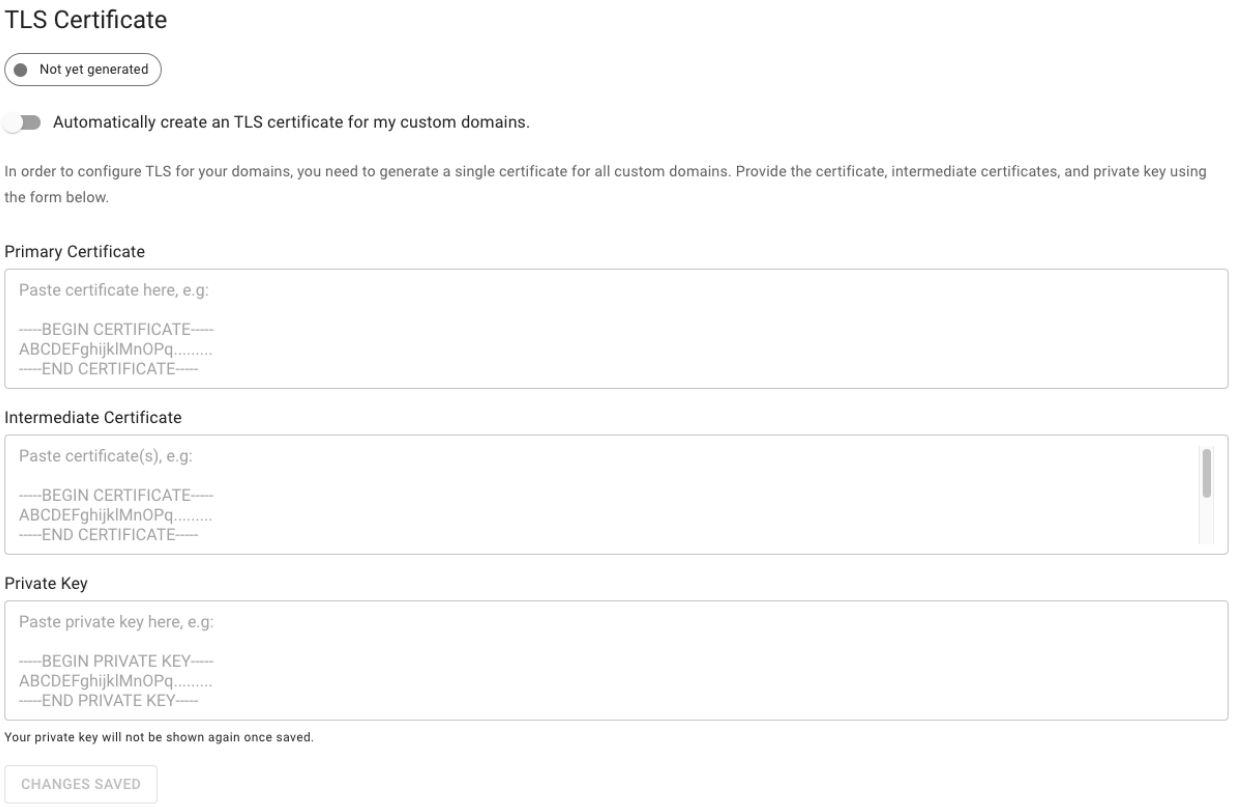
Uploading the certificate

To upload your SSL certificate, do the following:

1. Navigate to the *Settings* tab on your site:



2. Scroll to *TLS Certificate*.

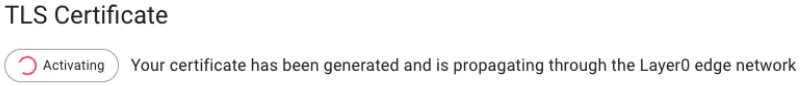


3. Toggle *Automatically create an TLS certificate for my custom domains* to the *on* position.
4. Copy the certificate, intermediate certificates, and the private key into the corresponding edit boxes.

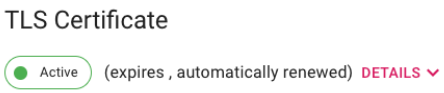
   > i
   >
   > The private key is non-public data and must not be shared with parties other than Edgio. Edgio stores your private key securely at rest. It is never shown in the developer console and only used to provision parts of the infrastructure that are used to terminate TLS connections.

5. Click *CHANGES SAVED*.

   The certificate's status becomes *Activating*:

   After the certificate is activated, its status becomes *Active*:

*Note: Certificate activation should take just a few minutes. If the status does not become Active within an hour, please contact **support**.*