Next.js

clerkMiddleware()

clerkMiddleware() | Next.js

The clerkMiddleware() helper integrates Clerk authentication into your Next.js application through Middleware. clerkMiddleware() is compatible with both the App and Pages routers.

Configure clerkMiddleware()

Create a middleware.ts file at the root of your project, or in your src/ directory if you have one.

Note

For more information about Middleware in Next.js, see the Next.js documentation.

middleware.ts

```
import { clerkMiddleware } from '@clerk/nextjs/server'

export default clerkMiddleware()

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params
'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

By default, clerkMiddleware will not protect any routes. All routes are public and you must opt-in to protection for routes.

createRouteMatcher()

createRouteMatcher() is a Clerk helper function that allows you to protect multiple routes. createRouteMatcher() accepts an array of routes and checks if the route the user is trying to visit matches one of the routes passed to it. The paths provided to this helper can be in the same format as the paths provided to the Next Middleware matcher.

The createRouteMatcher() helper returns a function that, if called with the req object from the Middleware, will return true if the user is trying to access a route that matches one of the routes passed to createRouteMatcher().

In the following example, createRouteMatcher() sets all /dashboard and /forum routes as protected routes.

```
const isProtectedRoute = createRouteMatcher(['/dashboard(.*)', '/forum(.*)'])
```

Protect routes

You can protect routes by checking either or both of the following:

User authentication status (user is signed in or out)

User authorization status (user has the required role or permission)

Protect routes based on user authentication status

You can protect routes based on user authentication status by checking if the user is signed in.

There are two methods that you can use:

Use auth.protect() if you want to redirect unauthenticated users to the sign-in route automatically.

Use auth().userId if you want more control over what your app does based on user authentication status.

auth.protect()

middleware.ts

```
import { clerkMiddleware, createRouteMatcher } from '@clerk/nextjs/server'

const isProtectedRoute = createRouteMatcher(['/dashboard(.*)', '/forum(.*)'])

export default clerkMiddleware(async (auth, req) => {
  if (isProtectedRoute(req)) await auth.protect()
})

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

auth().userId()

app/middleware.ts

```
import { clerkMiddleware, createRouteMatcher } from '@clerk/nextjs/server'

const isProtectedRoute = createRouteMatcher(['/dashboard(.*)', '/forum(.*)'])

export default clerkMiddleware(async (auth, req) => {
  const { userId, redirectToSignIn } = await auth()


  if (!userId && isProtectedRoute(req)) {
    // Add custom logic to run before redirecting


    return redirectToSignIn()
  }
})


export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

Protect routes based on user authorization status

You can protect routes based on user authorization status by checking if the user has the required roles or permissions.

There are two methods that you can use:

Use auth.protect() if you want Clerk to return a 404 if the user does not have the role or permission.

Use auth().has() if you want more control over what your app does based on the authorization status.

auth.protect()

middleware.ts

```
import { clerkMiddleware, createRouteMatcher } from '@clerk/nextjs/server'

const isProtectedRoute = createRouteMatcher(['/admin(.*)'])

export default clerkMiddleware(async (auth, req) => {
  const { has, redirectToSignIn } = await auth()
  // Restrict admin routes to users with specific permissions
  if (
    (isProtectedRoute(req) && !has({ permission: 'org:sys_memberships:manage' })) ||
    !has({ permission: 'org:sys_domains_manage' })
  ) {
    // Add logic to run if the user does not have the required permissions

    return redirectToSignIn()
  }
})

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|w
ebmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

auth().has()

middleware.ts

```
import { clerkMiddleware, createRouteMatcher } from '@clerk/nextjs/server'

const isProtectedRoute = createRouteMatcher(['/admin(.*)'])

export default clerkMiddleware(async (auth, req) => {
  const { has, redirectToSignIn } = await auth()
  // Restrict admin routes to users with specific permissions
  if (
    (isProtectedRoute(req) && !has({ permission: 'org:sys_memberships:manage' })) ||
    !has({ permission: 'org:sys_domains_manage' })
  ) {
    // Add logic to run if the user does not have the required permissions

    return redirectToSignIn()
  }
})

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

Protect multiple groups of routes

You can use more than one createRouteMatcher() in your application if you have two or more groups of routes.

The following example uses the has() method from the auth() helper.

middleware.ts

```
import { clerkMiddleware, createRouteMatcher } from '@clerk/nextjs/server'

const isTenantRoute = createRouteMatcher(['/organization-selector(.*)', '/orgid/(.*)'])

const isTenantAdminRoute = createRouteMatcher(['/orgId/(.*)/memberships',
'/orgId/(.*)/domain'])

export default clerkMiddleware(async (auth, req) => {
  // Restrict admin routes to users with specific permissions
  if (isTenantAdminRoute(req)) {
    await auth.protect((has) => {
      return (
        has({ permission: 'org:sys_memberships:manage' }) ||
        has({ permission: 'org:sys_domains_manage' })
      )
    })
  }
  // Restrict organization routes to signed in users
  if (isTenantRoute(req)) await auth.protect()
})
```

```
export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

Protect all routes

To protect all routes in your application and define specific routes as public, you can use any of the above methods and simply invert the if condition.

middleware.ts

```
import { clerkMiddleware, createRouteMatcher } from '@clerk/nextjs/server'

const isPublicRoute = createRouteMatcher(['/sign-in(.*)', '/sign-up(.*)'])

export default clerkMiddleware(async (auth, request) => {
  if (!isPublicRoute(request)) {
    await auth.protect()
  }
})

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

Debug your Middleware

If you are having issues getting your Middleware dialed in, or are trying to narrow down auth-related issues, you can use the debugging feature in clerkMiddleware(). Add { debug: true } to clerkMiddleware() and you will get debug logs in your terminal.

middleware.ts

```
import { clerkMiddleware } from '@clerk/nextjs/server'

export default clerkMiddleware(
  (auth, req) => {
    // Add your middleware checks
  },
  { debug: true },
)

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params

'/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

If you would like to set up debugging for your development environment only, you can use the process.env.NODE_ENV variable to conditionally enable debugging. For example, { debug: process.env.NODE_ENV === 'development' }.

Combine Middleware

You can combine other Middleware with Clerk's Middleware by returning the second Middleware from clerkMiddleware().

middleware.ts

```
import { clerkMiddleware, createRouteMatcher, redirectToSignIn } from '@clerk/nextjs/server'
import createMiddleware from 'next-intl/middleware'

import { AppConfig } from './utils/AppConfig'

const intlMiddleware = createMiddleware({
  locales: AppConfig.locales,
  localePrefix: AppConfig.localePrefix,
  defaultLocale: AppConfig.defaultLocale,
})

const isProtectedRoute = createRouteMatcher(['dashboard/(.*)'])

export default clerkMiddleware(async (auth, req) => {
  if (isProtectedRoute(req)) await auth.protect()
  return intlMiddleware(req)
})

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found in search params
    '/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)',
    // Always run for API routes
    '/(api|trpc)(.*)',
  ],
}
```

clerkMiddleware() options

The clerkMiddleware() function accepts an optional object. The following options are available:

audience?

string | string[]

A string or list of audiences. If passed, it is checked against the aud claim in the token.

authorizedParties?

string[]

An allowlist of origins to verify against, to protect your application from the subdomain cookie leaking attack.

For example: ['http://localhost:3000', 'https://example.com']

clockSkewInMs?

number

Specifies the allowed time difference (in milliseconds) between the Clerk server (which generates the token) and the clock of the user's application server when validating a token. Defaults to 5000 ms (5 seconds).

domain?

string

The domain used for satellites to inform Clerk where this application is deployed.

isSatellite?

boolean

When using Clerk's satellite feature, this should be set to true for secondary domains.

jwtKey

string

Used to verify the session token in a networkless manner. Supply the PEM public key from the API keys page -> Show JWT public key -> PEM Public Key section in the Clerk Dashboard. It's recommended to use the environment variable instead. For more information, refer to Manual JWT verification.

proxyUrl?

string

Specify the URL of the proxy, if using a proxy.

signInUrl?

string

An alternative sign in URL.

signUpUrl?

string

An alternative sign up URL.

publishableKey

string

The Clerk Publishable Key for your instance. This can be found on the API keys page in the Clerk Dashboard.

secretKey?

string

The Clerk Secret Key for your instance. This can be found on the API keys page in the Clerk Dashboard. The CLERK_ENCRYPTION_KEY environment variable must be set when providing secretKey as an option, refer to Dynamic keys.

It's also possible to dynamically set options based on the incoming request:

middleware.ts

```
import { clerkMiddleware } from '@clerk/nextjs/server'

export default clerkMiddleware(
  (auth, req) => {
    // Add your middleware checks
  },
  (req) => ({
    // Provide `domain` based on the request host
    domain: req.nextUrl.host,
  }),
)
```

Dynamic keys

Note

Dynamic keys are not accessible on the client-side.

The following options, known as "Dynamic Keys," are shared to the Next.js application server through clerkMiddleware, enabling access by server-side helpers like auth():

signUpUrl

signInUrl

secretKey

publishableKey

Dynamic keys are encrypted and shared during request time using a AES encryption algorithm. When providing a secretKey, the CLERK_ENCRYPTION_KEY environment variable is mandatory and used as the encryption key. If no secretKey is provided to clerkMiddleware, the encryption key defaults to CLERK_SECRET_KEY.

When providing CLERK_ENCRYPTION_KEY, it is recommended to use a 32-byte (256-bit), pseudorandom value. You can use openssl to generate a key:

terminal

```
openssl rand --hex 32
```

For multi-tenant applications, you can dynamically define Clerk keys depending on the incoming request. Here's an example:

middleware.ts

```
import { clerkMiddleware } from '@clerk/nextjs/server'

// You would typically fetch these keys from a external store or environment variables.
const tenantKeys = {
  tenant1: { publishableKey: 'pk_tenant1...', secretKey: 'sk_tenant1...' },
  tenant2: { publishableKey: 'pk_tenant2...', secretKey: 'sk_tenant2...' },
}

export default clerkMiddleware(
  (auth, req) => {
    // Add your middleware checks
  },
  (req) => {
    // Resolve tenant based on the request
    const tenant = getTenant(req)
    return tenantKeys[tenant]
  },
)
```