App Router setup with i18n routing

In order to use unique pathnames for every language that your app supports, `next-intl` can be used to handle the following routing setups:

1. Prefix-based routing (e.g. `/en/about`)
2. Domain-based routing (e.g. `en.example.com/about`)

In either case, `next-intl` integrates with the App Router by using a top-level `[locale]` dynamic segment that can be used to provide content in different languages.

Getting started

If you haven't done so already, create a Next.js app that uses the App Router and run:

```
npm install next-intl
```

Now, we're going to create the following file structure:

```
├── messages
|   ├── en.json
|   └── ...
├── next.config.ts
└── src
    ├── i18n
    |   ├── routing.ts
    |   ├── navigation.ts
    |   └── request.ts
    ├── middleware.ts
    └── app
        └── [locale]
            ├── layout.tsx
            └── page.tsx
```

In case you're migrating an existing app to `next-intl`, you'll typically move your existing pages into the `[locale]` folder as part of the setup.

Let's set up the files:

1 `messages/en.json`

Messages represent the translations that are available per language and can be provided either locally or loaded from a remote data source.

The simplest option is to add JSON files in your local project folder:

messages/en.json

```
{
  "HomePage": {
    "title": "Hello world!",
    "about": "Go to the about page"
  }
}
```

2 `next.config.ts`

Now, set up the plugin which creates an alias to provide a request-specific i18n configuration like your messages to Server Components—more on this in the following steps.

next.config.ts | next.config.js

next.config.js

```
const createNextIntlPlugin = require('next-intl/plugin');

const withNextIntl = createNextIntlPlugin();

/** @type {import('next').NextConfig} */
const nextConfig = {};

module.exports = withNextIntl(nextConfig);
```

3 `src/i18n/routing.ts`

We'll integrate with Next.js' routing in two places:

1. **Middleware**: Negotiates the locale and handles redirects & rewrites (e.g. `/` → `/en`)
2. **Navigation APIs**: Lightweight wrappers around Next.js' navigation APIs like `<Link />`

This enables you to work with pathnames like `/about`, while i18n aspects like language prefixes are handled behind the scenes.

To share the configuration between these two places, we'll set up `routing.ts`:

src/i18n/routing.ts

```ts
import {defineRouting} from 'next-intl/routing';

export const routing = defineRouting({
  // A list of all locales that are supported
  locales: ['en', 'de'],

  // Used when no locale matches
  defaultLocale: 'en'
});
```

Depending on your requirements, you may wish to customize your routing configuration later—but let's finish with the setup first.

4 `src/i18n/navigation.ts`

Once we have our routing configuration in place, we can use it to set up the navigation APIs.

src/i18n/navigation.ts

```ts
import {createNavigation} from 'next-intl/navigation';
import {routing} from './routing';

// Lightweight wrappers around Next.js' navigation
// APIs that consider the routing configuration
export const {Link, redirect, usePathname, useRouter, getPathname} =
  createNavigation(routing);
```

5 `src/middleware.ts`

Additionally, we can use our routing configuration to set up the middleware.

src/middleware.ts

```ts
import createMiddleware from 'next-intl/middleware';
import {routing} from './i18n/routing';

export default createMiddleware(routing);

export const config = {
  // Match all pathnames except for
  // - … if they start with `/api`, `/trpc`, `/_next` or `/_vercel`
  // - … the ones containing a dot (e.g. `favicon.ico`)
  matcher: '/((?!api|trpc|_next|_vercel|.*\\..*).*)'
};
```

> **How can I match pathnames that contain dots like `/users/jane.doe`?**

6 `src/i18n/request.ts`

When using features from `next-intl` in Server Components, the relevant configuration is read from a central module that is located at `i18n/request.ts` by convention. This configuration is scoped to the current request and can be used to provide messages and other options based on the user's locale.

src/i18n/request.ts

```ts
import {getRequestConfig} from 'next-intl/server';
import {hasLocale} from 'next-intl';
import {routing} from './routing';

export default getRequestConfig(async ({requestLocale}) => {
  // Typically corresponds to the `[locale]` segment
  const requested = await requestLocale;
  const locale = hasLocale(routing.locales, requested)
    ? requested
    : routing.defaultLocale;

  return {
    locale,
    messages: (await import(`../../messages/${locale}.json`)).default
  };
});
```

> Can I move this file somewhere else?

7 `src/app/[locale]/layout.tsx`

The `locale` that was matched by the middleware is available via the `locale` param and can be used to configure the document language. Additionally, we can use this place to pass configuration from `i18n/request.ts` to Client Components via `NextIntlClientProvider`.

app/[locale]/layout.tsx

```tsx
import {NextIntlClientProvider, hasLocale} from 'next-intl';
import {notFound} from 'next/navigation';
import {routing} from '@/i18n/routing';

export default async function LocaleLayout({
  children,
  params
}: {
  children: React.ReactNode;
  params: Promise<{locale: string}>;
}) {
  // Ensure that the incoming `locale` is valid
  const {locale} = await params;
  if (!hasLocale(routing.locales, locale)) {
    notFound();
  }

  return (
    <html lang={locale}>
      <body>
        <NextIntlClientProvider>{children}</NextIntlClientProvider>
      </body>
    </html>
  );
}
```

8 `src/app/[locale]/page.tsx`

Now you can use translations and other functionality from `next-intl` in your components:

app/[locale]/page.tsx

```
import {useTranslations} from 'next-intl';
import {Link} from '@/i18n/navigation';

export default function HomePage() {
  const t = useTranslations('HomePage');
  return (
    <div>
      <h1>{t('title')}</h1>
      <Link href="/about">{t('about')}</Link>
    </div>
  );
}
```

In case of async components, you can use the awaitable `getTranslations` function instead:

app/[locale]/page.tsx

```
import {getTranslations} from 'next-intl/server';

export default async function HomePage() {
  const t = await getTranslations('HomePage');
  return <h1>{t('title')}</h1>;
}
```

That's all it takes!

In case you ran into an issue, have a look at the App Router example to explore a working app.

💡 **Next steps:**

- Usage guide: Learn how to format messages, dates and times
- Routing: Set up localized pathnames, domain-based routing & more
- Workflows: Integrate deeply with TypeScript and other tools

Static rendering

When using the setup with i18n routing, `next-intl` will currently opt into dynamic rendering when APIs like `useTranslations` are used in Server Components. This is a

limitation that we aim to remove in the future, but as a stopgap solution, `next-intl` provides a temporary API that can be used to enable static rendering.

1 Add `generateStaticParams`

Since we are using a dynamic route segment for the `[locale]` param, we need to pass all possible values to Next.js via `generateStaticParams` so that the routes can be rendered at build time.

Depending on your needs, you can add `generateStaticParams` either to a layout or pages:

1. **Layout**: Enables static rendering for all pages within this layout (e.g. `app/[locale]/layout.tsx`)
2. **Individual pages**: Enables static rendering for a specific page (e.g. `app/[locale]/page.tsx`)

Example:

```
import {routing} from '@/i18n/routing';

export function generateStaticParams() {
  return routing.locales.map((locale) => ({locale}));
}
```

2 Add `setRequestLocale` to all relevant layouts and pages

`next-intl` provides an API that can be used to distribute the locale that is received via `params` in layouts and pages for usage in all Server Components that are rendered as part of the request.

app/[locale]/layout.tsx

```
import {setRequestLocale} from 'next-intl/server';
import {hasLocale} from 'next-intl';
import {notFound} from 'next/navigation';
import {routing} from '@/i18n/routing';

export default async function LocaleLayout({children, params}) {
  const {locale} = await params;
  if (!hasLocale(routing.locales, locale)) {
```

```
    notFound();
  }

  // Enable static rendering
  setRequestLocale(locale);

  return (
    // ...
  );
}
```

app/[locale]/page.tsx

```
import {use} from 'react';
import {setRequestLocale} from 'next-intl/server';
import {useTranslations} from 'next-intl';

export default function IndexPage({params}) {
  const {locale} = use(params);

  // Enable static rendering
  setRequestLocale(locale);

  // Once the request locale is set, you
  // can call hooks from `next-intl`
  const t = useTranslations('IndexPage');

  return (
    // ...
  );
}
```

Keep in mind that:

1. The locale that you pass to `setRequestLocale` should be validated (e.g. in your root layout).
2. You need to call this function in every page and every layout that you intend to enable static rendering for since Next.js can render layouts and pages independently.
3. `setRequestLocale` needs to be called before you invoke any functions from `next-intl` like `useTranslations` or `getMessages`.

> **How does setRequestLocale work?**

> **Why is this API necessary?**

3 Use the `locale` param in metadata

In addition to the rendering of your pages, also page metadata needs to qualify for static rendering.

To achieve this, you can forward the `locale` that you receive from Next.js via `params` to the awaitable functions from `next-intl`.

page.tsx

```tsx
import {getTranslations} from 'next-intl/server';

export async function generateMetadata({params}) {
  const {locale} = await params;
  const t = await getTranslations({locale, namespace: 'Metadata'});

  return {
    title: t('title')
  };
}
```

Last updated on April 7, 2025