Introduction

TypeScript is a JavaScript superset which extends the language to include type definitions allowing codebases to be statically checked for soundness. Gatsby provides an integrated experience out of the box, similar to an IDE. If you are new to TypeScript, adoption can *and should* be incremental. Since Gatsby natively supports JavaScript and TypeScript, you can change files from `.js` / `.jsx` to `.ts` / `.tsx` at any point to start adding types and gaining the benefits of a type system.

To see all of the types available and their generics look at our TypeScript definition file.

Initializing a new project with TypeScript

You can get started with TypeScript and Gatsby by using the CLI:

```
npm init gatsby
```

In the prompts, select TypeScript as your preferred language. You can also pass a `ts` flag to the above command to skip that question and use TypeScript:

```
npm init gatsby -ts
```

Usage in Gatsby

`PageProps`

```
import * as React from "react"
import type { PageProps } from "gatsby"

const IndexRoute = ({ path }: PageProps) => {
  return (
    <main>
      <h1>Path: {path}</h1>
    </main>
  )
}

export default IndexRoute
```

The example above uses the power of TypeScript, in combination with exported types from Gatsby (`PageProps`) to tell this code what props is. This can greatly improve your developer experience by letting your IDE show you what properties are injected by Gatsby.

`PageProps` can receive a couple of generics, most notably the `DataType` one. This way you can type the resulting `data` prop.

```
import * as React from "react"
import { graphql, PageProps } from "gatsby"

type DataProps = {
  site: {
    siteMetadata: {
      title: string
    }
  }
}

const IndexRoute = ({ data: { site } }: PageProps<DataProps>) => {
  return (
    <main>
      <h1>{site.siteMetadata.title}</h1>
    </main>
  )
}

export default IndexRoute

export const query = graphql`
  {
    site {
      siteMetadata {
        title
      }
    }
  }
`
```

`gatsby-browser.tsx` / `gatsby-ssr.tsx`

> *Support added in* `gatsby@4.8.0`

You can also write `gatsby-browser` and `gatsby-ssr` in TypeScript. You have the types `GatsbyBrowser` and `GatsbySSR` available to type your API functions. Here are two examples:

```
import * as React from "react"
import type { GatsbyBrowser } from "gatsby"

export const wrapPageElement: GatsbyBrowser["wrapPageElement"] = ({
  element,
}) => {
  return (
    <div>
      <h1>Hello World</h1>
      {element}
    </div>
  )
}
```

```
import * as React from "react"
import type { GatsbySSR } from "gatsby"
```

```
export const wrapPageElement: GatsbySSR["wrapPageElement"] = ({ element }) => {
  return (
    <div>
      <h1>Hello World</h1>
      {element}
    </div>
  )
}
```

`getServerData`

You can use `GetServerData`, `GetServerDataProps`, and `GetServerDataReturn` for [`getServerData`](getServerData).

```
import * as React from "react"
import type { GetServerDataProps, GetServerDataReturn } from "gatsby"

type ServerDataProps = {
  hello: string
}

const Page = () => <div>Hello World</div>
export default Page

export async function getServerData(
  props: GetServerDataProps
): GetServerDataReturn<ServerDataProps> {
  return {
    status: 200,
    headers: {},
    props: {
      hello: "world",
    },
  }
}
```

If you're using an anonymous function, you can also use the shorthand `GetServerData` type like this:

```
const getServerData: GetServerData<ServerDataProps> = async props => {
  // your function body
}
```

`gatsby-config.ts`

> *Support added in `gatsby@4.9.0`*

You can import the type `GatsbyConfig` to type your config object. **Please note:** There are currently no type hints for `plugins` and you'll need to check the current limitations and see if they apply to your `gatsby-config.ts` file.

```
import type { GatsbyConfig } from "gatsby"

const config: GatsbyConfig = {
  siteMetadata: {
    title: "Your Title",
  },
  plugins: [],
}

export default config
```

Read the Gatsby Config API documentation to learn more about its different options.

`gatsby-node.ts`

> *Support added in* `gatsby@4.9.0`

You can import the type `GatsbyNode` to type your APIs by accessing keys on `GatsbyNode`, e.g. `GatsbyNode["sourceNodes"]`. **Please note:** You'll need to check the current limitations and see if they apply to your `gatsby-node.ts` file.

```
import type { GatsbyNode } from "gatsby"

type Person = {
  id: number
  name: string
  age: number
}

export const sourceNodes: GatsbyNode["sourceNodes"] = async ({
  actions,
  createNodeId,
  createContentDigest,
}) => {
  const { createNode } = actions

  const data = await getSomeData()

  data.forEach((person: Person) => {
    const node = {
      ...person,
      parent: null,
      children: [],
      id: createNodeId(`person__${person.id}`),
      internal: {
        type: "Person",
        content: JSON.stringify(person),
        contentDigest: createContentDigest(person),
      },
    }
    createNode(node)
  })
}
```

Read the Gatsby Node APIs documentation to learn more about its different APIs.

Local Plugins

> *Support added in* `gatsby@4.9.0`

All the files mentioned above can also be written and used inside a local plugin.

`tsconfig.json`

Essentially, the `tsconfig.json` file is used in tools external to Gatsby e.g Testing Frameworks like Jest, Code editors and Linting libraries like EsLint to enable them handle TypeScript correctly. You can use the `tsconfig.json` from our gatsby-minimal-starter-ts.

Type Hinting in JS Files

You can still take advantage of type hinting in JavaScript files with JSdoc by importing types directly from Gatsby. You need to use a text exitor that supports those type hints.

Usage in `gatsby-config.js`

```
// @ts-check

/**
 * @type {import('gatsby').GatsbyConfig}
 */
const gatsbyConfig = {}

module.exports = gatsbyConfig
```

Usage in `gatsby-node.js`

```
// @ts-check

/**
 * @type {import('gatsby').GatsbyNode['createPages']}
 */
exports.createPages = () => {}
```

Styling

vanilla-extract helps you write type-safe, locally scoped classes, variables and themes. It's a great solution when it comes to styling in your TypeScript project. To use vanilla-extract, select it as your preferred styling solution when initializing your project with `npm init gatsby`. You can also manually setup your project through gatsby-plugin-vanilla-extract or use the vanilla-extract example.

Migrating to TypeScript

Gatsby natively supports JavaScript and TypeScript, you can change files from `.js` / `.jsx` to `.ts` / `tsx` at any point to start adding types and gaining the benefits of a type system. But you'll need to do a bit more to be able use Typescipt in `gatsby-*` files:

- Run `gatsby clean` to remove any old artifacts
- Convert your `.js` / `.jsx` files to `.ts/.tsx`
- Install `@types/node`, `@types/react`, `@types/react-dom`, `typescript` as `devDependencies`
- Add a `tsconfig.json` file using `npx tsc --init` or use the one from [gatsby-minimal-starter-ts]()
- Rename `gatsby-*` files:
  - `gatsby-node.js` to `gatsby-node.ts`
  - `gatsby-config.js` to `gatsby-config.ts`
  - `gatsby-browser.js` to `gatsby-browser.tsx`
  - `gatsby-ssr.js` to `gatsby-ssr.tsx`
- Address any of the [current limitations]()

If you've used other ways of using TypeScript in the past, you'll also want to read [migrating away from old workarounds]().

Current limitations

There are some limitations currently that you need to be aware of. We'll do our best to mitigate them in our code, through contributions to upstream dependencies, and updates to our documentation.

Parcel TypeScript features

Parcel is used for the compilation and it currently has [limitations on TypeScript features](), namely:

- No support for `baseUrl` or `paths` inside `tsconfig.json`
- It implicitly enables the [`isolatedModules`]() option by default

`__dirname`

You can't use `__dirname` and `__filename` in your files. You'll need to replace these instances with a `path.resolve` call. Example diff for a `gatsby-config` file:

```diff
+ import path from "path"

const config = {
  plugins: [
    {
      resolve: `gatsby-source-filesystem`,
      options: {
        name: `your-name`,
+       path: path.resolve(`some/folder`),
-       path: `${__dirname}/some/folder`,
      },
    },
  ]
}
```

```
export default config
```

Progress on this is tracked in Parcel #7611.

`require.resolve`

You can't use `require.resolve` in your files. You'll need to replace these instances with a `path.resolve` call. Example diff for a `gatsby-node` file:

```diff
+ import path from "path"

+ const template = path.resolve(`src/templates/template.tsx`)
- const template = require.resolve(`./src/templates/template.tsx`)
```

Progress on this is tracked in Parcel #6925.

Other

- Workspaces (e.g. Yarn) are not supported. We'll add documentation on how to use this feature inside a workspace soon.
- When changing `siteMetadata` in `gatsby-config` no hot-reloading will occur during `gatsby develop`. A restart is needed at the moment.

Other Resources

TypeScript integration for pages is supported through automatically including `gatsby-plugin-typescript`. Visit that link to see configuration options and limitations of this setup.

If you are new to TypeScript, check out these other resources to learn more:

- TypeScript Documentation
- TypeScript Playground (Try it out!)