Gatsby Functions help you build Express-like backends without running servers.

Functions are generally available in sites running Gatsby 3.7 and above.

Introduction

JavaScript and TypeScript files in `src/api/*` are mapped to function routes like files in `src/pages/*` become pages. So `src/api` is a reserved directory for Gatsby. Gatsby by default ignores test files (e.g. `hello-world.test.js`) and dotfiles (e.g. `.prettierrc.js`).

For example, the following Function is run when you visit the URL `/api/hello-world`

```
export default function handler(req, res) {
  res.status(200).json({ hello: `world` })
}
```

A Function file must export a single function that takes two parameters:

- `req`: Node's http request object with some automatically parsed data
- `res`: Node's http response object with some extra helper functions

Dynamic routing is supported for creating REST-ful APIs and other uses cases

- `/api/users` => `src/api/users/index.js`
- `/api/users/23` => `src/api/users/[id].js`

Learn more about dynamic routes.

Typescript

Functions can be written in JavaScript or Typescript.

```
import { GatsbyFunctionRequest, GatsbyFunctionResponse } from "gatsby"

interface ContactBody {
  message: string
}

export default function handler(
  req: GatsbyFunctionRequest<ContactBody>,
  res: GatsbyFunctionResponse
) {
  res.send({ title: `I am TYPESCRIPT`, message: req.body.message })
}
```

Common data formats are automatically parsed

Query strings and common body content types are automatically parsed and available at `req.query` and `req.body`

Read more about supported data formats.

```
export default function contactFormHandler(req, res) {
  // "req.body" contains the data from a contact form
}
```

Respond to HTTP Methods

Sometimes you want to respond differently to GETs vs. POSTs or only respond to one method.

```
export default function handler(req, res) {
  if (req.method === `POST`) {
    res.send(`I am POST`)
  } else {
    // Handle other methods or return error
  }
}
```

Headers

Only HTTP headers prefixed with `x-gatsby-` are passed into your functions.

Environment variables

Site environment variables are used to pass secrets and environment-specific configuration to Functions.

```
import fetch from "node-fetch"

export default async function postNewPersonHandler(req, res) {
  // POST data to an authenticated API
  const url = "https://example.com/people"

  const headers = {
    "Content-Type": "application/json",
    Authorization: `Bearer ${process.env.CLIENT_TOKEN}`,
  }

  const data = {
    name: req.body.name,
    occupation: req.body.occupation,
    age: req.body.age,
  }

  try {
    const result = await fetch(url, {
      method: "POST",
      headers: headers,
      body: data,
    }).then(res => {
      return res.json()
    })
```

```
    res.json(result)
  } catch (error) {
    res.status(500).send(error)
  }
}
```

Forms

Forms and Functions are often used together. For a working example you can play with locally, see the form example. The Forms doc page is a gentle introduction for building forms in React. Below is sample code for a very simple form that submits to a function that you can use as a basis for building out forms in Gatsby.

```
export default function formHandler(req, res) {
  // req.body has the form values
  console.log(req.body)

  // Here is where you would validate the form values and
  // do any other actions with it you need (e.g. save it somewhere or
  // trigger an action for the user).
  //
  // e.g.

  if (!req.body.name) {
    return res.status(422).json("Name field is required")
  }

  return res.json(`OK`)
}
```

```
import * as React from "react"

export default function FormPage() {
  const [value, setValue] = React.useState({})
  const [serverResponse, setServerResponse] = React.useState(``)

  // Listen to form changes and save them.
  function handleChange(e) {
    value[e.target.id] = e.target.value
    setServerResponse(``)
    setValue({ ...value })
  }

  // When the form is submitted, send the form values
  // to our function for processing.
  async function onSubmit(e) {
    e.preventDefault()
    const response = await window
      .fetch(`/api/form`, {
        method: `POST`,
```

```
        headers: {
          "content-type": "application/json",
        },
        body: JSON.stringify(value),
      })
      .then(res => res.json())

    setServerResponse(response)
  }

  return (
    <div>
      <div>Server response: {serverResponse}</div>
      <form onSubmit={onSubmit} method="POST" action="/api/form">
        <label htmlFor="name">Name:</label>
        <input
          type="text"
          id="name"
          value={value[`name`] || ``}
          onChange={handleChange}
        />
        <input type="submit" />
      </form>
    </div>
  )
}
```

Functions in plugins and themes

Plugins and themes can ship functions! This is powerful as it lets you pair frontend code with backend code. For example, if you built a plugin for an authorization service that includes a login component, you could ship alongside the component, a serverless function the component can use to connect to the remote API.

Namespacing

Plugin/theme functions work exactly the same as normal functions except their routes must be created under the plugin's namespace e.g. `${PLUGIN_ROOT}/src/api/{pluginName}/my-plugin-function.js`.

Shadowing with functions works similar to how shadowing works in general. You can shadow a plugin/theme function by copying the file from the plugin/theme's `src` tree into your site's `src` tree. For example, to shadow the `/gatsby-plugin-cool/do-something` function from the `gatsby-plugin-cool` plugin, you'd copy `node_modules/gatsby-plugin-cool/src/api/gatsby-plugin-cool/do-something.js` to `src/api/gatsby-plugin-cool/do-something.js`. From there, you can overwrite the implementation of the `/do-something` function as you normally would.

Limitations

- Bundling in native dependencies is not supported at the moment