Configuring Plugin Usage with Plugin Options

Plugins loaded into a Gatsby site can have options passed in to customize how a plugin operates.

This guide refers to creating plugins, if you are looking for general information on using options with plugins refer to "Using a Plugin in Your Site". If you are looking for options of a specific plugin, refer to its README.

Where to access plugin options

A Gatsby plugin with options included makes those options available in the second argument of Gatsby Node, Browser, and SSR APIs. Consider the following `gatsby-config.js` with a plugin called `gatsby-plugin-console-log`:

```
module.exports = {
  plugins: [
    {
      resolve: `gatsby-plugin-console-log`,
      options: { optionA: true, optionB: false, message: "Hello world" },
    },
  ],
}
```

With the `optionA`, `optionB`, and `message` options passed into the plugin, the code for `gatsby-plugin-console-log` is able to access the values `true`, `false`, and `"Hello world"` by their keys.

For example, `gatsby-plugin-console-log` can access the `message` in order to log its value to the console inside of the `onPreInit` API:

```
exports.onPreInit = (_, pluginOptions) => {
  console.log(
    `logging: "${pluginOptions.message}" to the console` // highlight-line
  )
}
```

The code above is called when `gatsby develop` or `gatsby build` is run. It takes the `message` from the `options` object in the config and logs it from `pluginOptions.message` when the `onPreInit` method is called.

The second argument passed into the function is where the options are held.

Like arguments in any JavaScript function, you can use a different (more specific) name like `themeOptions` if you are building a plugin that will be used as a theme.

What can be passed in as options

Any JavaScript data type can be passed in as an option.

The following table lists possible options values and an example plugin that makes use of them.

Data Type	Sample Value	Example Plugin
Boolean	`true`	`gatsby-plugin-sharp`
String	`/src/data/`	`gatsby-source-filesystem`
Array	`["/about-us/", "/projects/*"]`	`gatsby-plugin-offline`
Object	`{ default: "./src/layout.js" }`	`gatsby-plugin-mdx`

Note: Themes (which are a type of plugin) are able to receive options from a site's `gatsby-config.js` to be used in its `gatsby-config.js` in order to allow themes to be composed together. This is done by exporting the `gatsby-config.js` as a function instead of an object. You can see an example of this in the `gatsby-theme-blog` and `gatsby-theme-blog-core` repositories. Plugins are not capable of this functionality.

How to validate plugin options

To help users configure plugins correctly, a plugin can optionally define a schema to enforce a type for each option. Gatsby will validate that the options users pass match the schema to help them correctly set up their site.

How to define an options schema

You should use the `pluginOptionsSchema` API to define your plugins' options schema. It gets passed an instance of Joi, which you use to return a `Joi.object` schema for the options you expect users to pass.

For example, imagine you were creating a plugin called `gatsby-plugin-console-log`. You decide you want users to configure your plugin using the following options:

```
module.exports = {
  plugins: [
    {
```

```
      resolve: `gatsby-plugin-console-log`,
      options: {
        optionA: true,
        message: "Hello world",
        optionB: false, // Optional.
      },
    },
  ],
}
```

You want users to pass in a boolean to `optionA` and a string to `message`, and they can optionally pass a boolean to `optionB`. To enforce these rules, you would create the following `pluginOptionsSchema`:

```
exports.pluginOptionsSchema = ({ Joi }) => {
  return Joi.object({
    optionA: Joi.boolean().required().description(`Enables optionA.`),
    message: Joi.string()
      .required()
      .description(`The message logged to the console.`),
    optionB: Joi.boolean().description(`Enables optionB.`),
  })
}
```

If users pass options that do not match the schema, the validation will show an error when they run `gatsby develop` and prompt them to fix their configuration.

For example, if an integer is passed into `message` (which is marked as a required string) this message would be shown:

```
 ERROR #11331  PLUGIN

Invalid plugin options for "gatsby-plugin-console-log":

- "message" must be a string
```

Best practices for option schemas

The Joi API documentation is a great reference to use while working on a `pluginOptionsSchema`, as it shows all the available types and methods.

Here are some specific Joi best practices for `pluginOptionsSchema`:

- Add descriptions
- Set default options
- Validate external access where necessary
- Add custom error messages where useful
- Deprecate options in a major version release rather than removing them

Add descriptions Make sure that every option and field has a `.description()` explaining its purpose. This is helpful for documentation as users can look at the schema and understand all the options. There might also be tooling in the future that auto-generates plugin option documentation from the schema.

Set default options You can use the `.default()` method to set a default value for an option. For example, in the `gatsby-plugin-console-log` plugin above, you could have the `message` option default to `"default message"` if a user does not pass their own `message` value:

```
exports.pluginOptionsSchema = ({ Joi }) => {
  return Joi.object({
    optionA: Joi.boolean().required().description(`Enables optionA.`),
    message: Joi.string()
      .default(`default message`) // highlight-line
      .description(`The message logged to the console.`),
    optionB: Joi.boolean().description(`Enables optionB.`),
  })
}
```

Accessing `pluginOptions.message` would then log `"default message"` in all plugin APIs if the user does not supply their own value.

Validate external access Some plugins (particularly source plugins) query external APIs. With the `.external()` method, you can asynchronously validate that the user has access to the API, providing a better experience if they pass invalid secrets.

For example, this is how the Contentful source plugin might validate that the user has access to the space they are trying to query:

```
exports.pluginOptionsSchema = ({ Joi }) => {
  return Joi.object({
    accessToken: Joi.string().required(),
    spaceId: Joi.string().required(),
    // ...more options here...
  }).external(async pluginOptions => {
    try {
      await contentful
        .createClient({
          space: pluginOptions.spaceId,
          accessToken: pluginOptions.accessToken,
        })
        .getSpace()
    } catch (err) {
      throw new Error(
```

```
        `Cannot access Contentful space "${pluginOptions.spaceId}" with the provided access
      )
    }
  })
}
```

Add custom error messages Sometimes you might want to provide more detailed error messages when validation fails for a specific field. Joi provides a `.messages()` method which lets you override error messages for specific error types (e.g. `"any.required"` when a `.required()` call fails).

For example, in the `gatsby-plugin-console-log` plugin above, this is how you would provide a custom error message if users do not specify `optionA`:

```
exports.pluginOptionsSchema = ({ Joi }) => {
  return Joi.object({
    optionA: Joi.boolean()
      .required()
      .description(`Enables optionA.`)
      // highlight-start
      .messages({
        // Override the error message if the .required() call fails
        "any.required": `"optionA" needs to be specified to true or false. Get the correct v
      }),
    // highlight-end
    message: Joi.string()
      .default(`default message`)
      .description(`The message logged to the console.`),
    optionB: Joi.boolean().description(`Enables optionB.`),
  })
}
```

Deprecating options While you can simply remove options from the schema in major versions, that causes cryptic error messages for users upgrading with existing configuration. Instead, deprecate them using the `.forbidden()` method in a major version release. Then, add a custom error message explaining how users should upgrade the functionality using `.messages()`.

For example:

```
  return Joi.object({
    optionA: Joi.boolean()
      .required()
      .description(`Enables optionA.`)
      // highlight-start
      .forbidden()
      .messages({
```

```
        // Override the error message if the .forbidden() call fails
        "any.unknown": `"optionA" is no longer supported. Use "optionB" instead by setting i
      }),
      // highlight-end
    message: Joi.string()
      .default(`default message`)
      .description(`The message logged to the console.`),
    optionB: Joi.boolean().description(`Enables optionB.`),
  })
}
```

Unit testing an options schema

To verify that a `pluginOptionsSchema` behaves as expected, unit test it with different configurations using the `gatsby-plugin-utils` package.

1. Add the `gatsby-plugin-utils` package to your site:

   ```
   npm install --dev gatsby-plugin-utils
   ```

2. Use the `testPluginOptionsSchema` function exported from the package in your test file. It takes two parameters, the plugin's actual Joi schema and an example options object to test. It returns an object with an `isValid` boolean, which will be true or false based on whether or not the options object fits the actual Joi schema, and an `errors` array, which will contain the error messages if the validation failed.

For example, with Jest, your tests might look something like this:

```
// This is an example using Jest (https://jestjs.io/)
import { testPluginOptionsSchema } from "gatsby-plugin-utils"
import { pluginOptionsSchema } from "../gatsby-node"

describe(`pluginOptionsSchema`, () => {
  it(`should invalidate incorrect options`, async () => {
    const options = {
      optionA: undefined, // Should be a boolean
      message: 123, // Should be a string
      optionB: `not a boolean`, // Should be a boolean
    }
    const { isValid, errors } = await testPluginOptionsSchema(
      pluginOptionsSchema,
      options
    )

    expect(isValid).toBe(false)
    expect(errors).toEqual([
      `"optionA" is required`,
```

```
      `"message" must be a string`,
      `"optionB" must be a boolean`,
    ])
  })

  it(`should validate correct options`, async () => {
    const options = {
      optionA: false,
      message: "string",
      optionB: true,
    }
    const { isValid, errors } = await testPluginOptionsSchema(
      pluginOptionsSchema,
      options
    )

    expect(isValid).toBe(true)
    expect(errors).toEqual([])
  })
})
```

Additional resources

- Example Gatsby site using plugin options with a local plugin
- Joi API documentation
- `pluginOptionsSchema` for the Contentful source plugin
- `pluginOptionsSchema` for the Kontent source plugin