Recipes

These docs are intended to serve as an introduction to Recipes, its motivations, and how they work internally.

TODO

- ☐ Define context in resources
- ☐ Move recipe asides on tech debt into issues

Table of contents

- Related
  - Alternatives considered
  - Inspiration
  - Resources
- Glossary

Introduction

Recipes is an "infrastructure as code" system that lets users automatically manage and provision the technology stack for their Gatsby site/app through code rather than manual processes.

It's powered by React and MDX. A useful analogy is "React Native for Infrastructure". It's declarative yet also allows for programmatic escape hatches and conditions via JSX.

Recipes also provides a read/write API for Desktop/Admin to build low-code tooling on top of Gatsby and its integrated services.

Goals

- Make your first 10 minutes using Gatsby feel magical
- Provide read/write API layer for Admin & Desktop
- Simplify installing and configuring Gatsby plugins
- Replace starters
- Solve secrets management
- Dramatically simplify provisioning and evolving more complex Gatsby "stacks"

Example Recipe

Below is an example Recipe that creates a hello world Markdown file.

```
# Create a file

---

Creates a "hello, world" file

<File name="hello.md" content="# Hello, world!" />

---

That's it!
```

This example will be used throughout this document as an illustration tool.

Setting up a development environment

First follow the instructions on setting up a local Gatsby dev environment.

Running the unit tests

```
yarn jest packages/gatsby-recipes
```

Testing and extending providers/resources

If you want to fix a bug in a resource or extend it in some way, typically you'll be working against the tests for that resource.

In your terminal, start a jest watch process against the resource you're working on e.g. for GatsbyPlugin:

```
yarn jest packages/gatsby-recipes --testPathPattern "providers" --watch
```

Using gatsby-dev-cli

You can create test recipes that you run in a test site. You'll need to use `gatsby-dev-cli` for this..

One note, as you'll be testing changes to the Gatsby CLI — instead of running the global gatsby-cli package (i.e. what you'd run by typing `gatsby`, you'll want to run the version copied over by `gatsby-dev-cli` by running `./node_modules/.bin/gatsby`.

When debugging the CLI, you may run into errors without stacktraces. In order to work around that, you can use the node inspector:

```
DEBUG=true node --inspect-brk ./node_modules/.bin/gatsby recipes ./test.mdx
```

Then, open up Chrome and click the node icon in dev tools.

To see log output from the Recipes graphql server, start the Recipes API in one terminal `node node_modules/gatsby-recipes/dist/graphql-server/server.js` and then in another terminal run your recipe with `RECIPES_DEV_MODE=true` set as an env variable.

Running with Gatsby Admin

Make sure all packages are built:

```
yarn bootstrap
```

Then run Gatsby Admin directly:

```
yarn workspace gatsby-admin run develop
```

Debugging your development environment

It's possible for your prior development GraphQL server to get caught in a "zombie" state when it hangs. If it appears that changes to the API aren't being reflected check your running processes to see if it's hanging with `ps aux`.

Architecture

In Recipes there's a notion of a *"client"* and the *"backend"*. The backend handles the running of a Recipe (for both plan and apply). They communicate over a GraphQL API.

GraphQL API

Recipes is intended to have two primary entrypoints, the Gatsby CLI and Gatsby Admin/Desktop. It follows a client/server model where the client sends a recipe's source code to the server via a GraphQL API. The server sends back a "plan".

The client receives the plan and uses that data to render a summary. The user can then opt to "apply plan" which will install the recipe on the current environment.

All clients (CLI, GUI, Admin) use `urql` to communicate with the API in order to request data, send mutations, or subscribe to updates when running a recipe.

The API itself can be run and used *outside* of Recipes as well, which is what Gatsby Admin does.

Schema If you're using `gatsby-dev-cli` in a project, you can connect directly to GraphiQL to explore the API endpoints. This will typically run at http://localhost:50400/graphql unless the server was automatically invoked by running `gatsby develop`.

Since GraphiQL documents itself, we won't go into detail.

Subscriptions The GraphQL API has an `operation` subscription. When a recipe is selected by the user an `operation` subscription is created and the state machine is invoked.

The server sends updates on state transitions (though some are ignored) and the client renders new UI based on the current state. If/when the user decides to apply the recipe an event is sent. If the user adds inputs (if they're used) an `INPUT_ADDED` event is also sent back to the server.

When an input is received the server re-runs the recipe rendering and then emits back the updated plan.

The renderer can be thought of as a runtime loop on both the client and the server, and events are emitted back and forth.

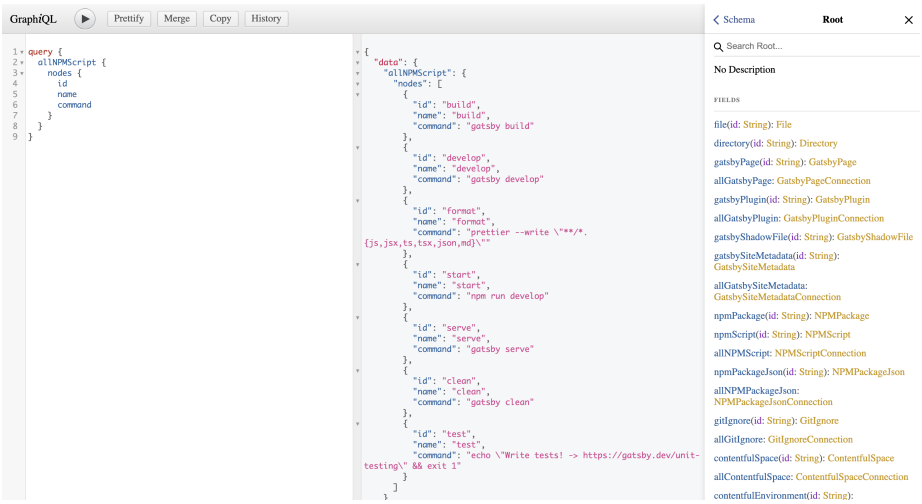

Figure 1: image

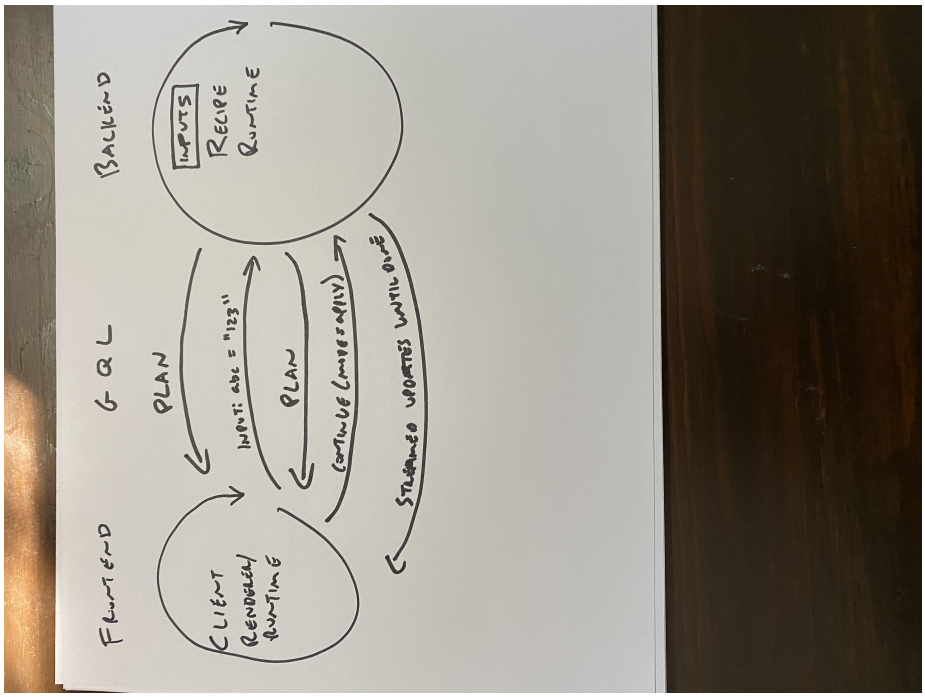

Figure 2: IMG_2969

Generating types Each resource defines its own schema which is what we use to generate GraphQL types for the API. We use a forked version of `joi-to-graphql` to do this for us. It's a little bit clunky because the shape that the library returns is a bit different than what we want for a code-first GraphQL definition so we have to massage it into place.

An aside on generating types This is a location of the code base that has some tech debt. Firstly, Recipes is running a pretty old version of `Joi` and can't update until the `joi-to-graphl` library which we've internally forked has been updated. It uses a lot of internal `Joi` APIs so it's not super straightforward to update.

The `joi-to-graphql` library was forked in the first place to allow us to handle more types from Joi that we need in resources. The library itself isn't maintained, at some point we might want to rewrite it.

See Joi update PR for more context →

State machine

The recipe state machine is specific to Recipes and handles the flow control of logic. When a recipe is sent to the backend the GraphQL server initializes a state machine and when the state changes, events are sent back to the client as part of the GraphQL subscription.

States

- `resolveRecipe`
- `parseRecipe`
- `validateSteps`: Ensure that the parsed recipe appears valid
- `createPlan`
- `presentPlan`: Emit the plan to the user
- `applyPlan`
- `done`: That's all folks

Events

- `CONTINUE`: Apply the plan
- `INPUT_ADDED`: Update the input in context, rerun `presentPlan`
- `TICK`: For tracking long running plans and update the client
- `RESET`: Reset `TICK` count

Actions

- `addResourcesToContext`: As updates happen to a plan being applied this global object is updated and set to the client so it can update

Recipe resolution

Recipes can come from a few places, and the recipe resolution step in the state machine handles this.

- **File system**: `gatsby recipes ./my-local-file.mdx`
- **Official recipes**: `gatsby recipes theme-ui`
- **Url**: `gatsby recipes https://gist.github.com/123abc`

Parser

The parser receives the MDX source code of a recipe and parses it with the MDX (v2) parser. The recipe is then partitioned on all `thematicBreak` nodes (`---`). Each of these partitioned chunks are a *"step"*, with the first step serving as the recipe introduction.

Each component in the MDX document is given a `uuid` so that its state can be tracked, and each step is wrapped up in a `Step` component to provide any step context in case components might want access to it during the render process.

Exports are also plucked and are exposed to rendering for each step since they're technically global to the entire document. Exports allow folks to instantiate variables and even conditional components to be used:

```
# Create a file

---

Creates a "hello, world" file

export const fileName = "hello.md"

<File name={fileName} content="# Hello, world!" />

---

That's it!
```

Input MDX

```
# Create a file

---

Creates a "hello, world" file

<File name="hello.md" content="# Hello, world!" />

---
```

```
That's it!
```

Output MDX

```
<RecipeIntroduction># Create a file</RecipeIntroduction>

<RecipeStep step={1} total={2}>
Creates a "hello, world" file

<File name="hello.md" content="# Hello, world!" _uuid="123abc" />
</RecipeStep>

<RecipeStep step={2} total={2}>
  That's it!
</RecipeStep>
```

Parser return object The parser itself returns different variations of the recipe source code so that it can be displayed on the client in different ways.

- `exports`: So that they can be exposed for each step when rendered
- `stepsAsMdx`: Used for the step by step display
- `stepsAsJs`: So that babel doesn't have to run on the client (WIP)
- `ast`: This isn't used and can probably be removed
- `recipe`: Full document

Renderer

When a recipe plan is created or applied, it is rendered using a custom React reconciler. It's a full-fledged React runtime that renders to a JavaScript object that's transformed into a plan and sent back to the client.

In order to do this the original recipe MDX source code undergoes a few transformations:

- **parsed**
- **transformed** with Babel so it can be inline evaluated
- **evaluated** with `new Function` and the necessary scope

Recipes source code can contain nested resources and all resources are asynchronous by default. In order to work with this all resources are rendered with React Suspense and then an emitter is returned which emits events as resources are diffed and/or applied.

Reconciler

Input

```
<File path="red.js" content="red!">
  <File path="blue.js" content="blue!" />
</File>
```

Output

```
const result = [
  {
    resourceName: "File",
    resourceDefinitions: {
      content: "red!",
      path: "red.js",
    },
    currentState: "",
    describe: "Write red.js",
    diff: "OMITTED",
    newState: "red!",
    resourceChildren: [
      {
        resourceName: "File",
        resourceDefinitions: {
          content: "blue!",
          path: "blue.js",
        },
        currentState: "",
        describe: "Write blue.js",
        diff: "OMITTED",
        newState: "blue!",
      },
    ],
  },
]
```

Read more about the reconciler motivation →

Recipe components The resource handling with Suspense is one of the weirder parts of the codebase because Suspense can be a bit clunky to deal with in this circumstance.

When a resource, like `File`, is rendered it throws a custom promise that wraps the resource and its props/context are forwarded directly to the `plan` or `create` call in `providers/fs/file`.

In order to achieve this we wrap each resource that we generate from the providers and wrap that in Suspense.

```
const Resource = props => (
  <Suspense fallback={<p>Reading File...</p>}>
```

```
    <ResourceComponent _resourceName="File" {...props} />
  </Suspense>
)
```

The `ResourceComponent` looks like so (details omitted for brevity):

```
const ResourceComponent = ({
  _resourceName: Resource,
  _uuid,
  _type,
  children,
  ...props
}) => {
  /* ... */
  const resourceData = handleResource(
    Resource,
    {
      ...parentResourceContext,
      root: process.cwd(),
      _uuid,
      mode,
      resultCache,
      inFlightCache,
      blockedResources,
      queue,
    },
    props
  )

  return (
    <ParentResourceProvider
      data={
        {
          /* ... */
        }
      }
    >
      <Resource>
        {JSON.stringify({
          ...resourceData,
          _props: props,
          _stepMetadata: step,
          _uuid,
          _type,
        })}
        {children}
      </Resource>
```

```
    </ParentResourceProvider>
  )
}
```

It's wrapped in context that passes along any parent context and inputs which might affect the default props.

`handleResource` determines what mode we're in (plan vs apply) and then uses that to make the asynchronous calls, handle things like caching, and then eventually returns a JS object with the result from the resource call.

This is also where the notion of *"context"* comes into play with the resource. When rendered context is set up and then passed down in React-land. Right now we currently hardcode the project root and pass in other data like step and any parent resource information.

The results from `handleResource`, when the promise is resolved, are then serialized as a JSON string that the custom reconciler injects as text into the *"Recipes VDOM"* that's a JSON object.

Server renderer The server renderer is different from the client renderer because it uses the custom reconciler and once all resources have been rendered sits in an idle state until an input is received or the mode changes.

Client renderer There are two primary clients, the CLI and GUI, which are used to communicate with the GraphQL API. The API returns the current plan and transformed source MDX.

They render directly to DOM and makes aspects like input support render live on the page. This is also why the client has to emit input changes because those are added to context on the server renderer and that gets flushed out into a single plan when the event is received.

They leverage the aformentioned GraphQL API and render the UI based on what makes sense.

An aside on the clients This is probably the area where the code is the sloppiest and also the most difficult to untangle. It's mostly junky prototype code that's been layered on a few time. Though, it's generally mostly React that needs to be unwound, the CLI can be difficult to develop with sometimes because errors are lost.

Also, we need to ensure that the clients don't need to transform the JS, this is something the backend/parser should handle so that babel can live solely on the server side.

CLI The CLI uses `ink` to render React code to the terminal. It has a plan mode and an apply mode via `--install` much like Terraform.

GUI The GUI code is currently being moved into Gatsby Admin. It operates similarly to the CLI but renders in the browser and will have proper support for inputs and able to use `gatsby-interface`.

Render in apply mode The backend renderer defaults to *"plan"* mode when rendering which means it returns the diff between the current state and intended state rather than updating the current environment.

When the renderer is told to *apply* it will call `create` on the resource rather than `update`. This will actively update the environment and then emit its status back to the client.

Providers and resources

Providers and resources are the bread and butter of Recipes. This is where we anticipate most of the development to happen when the internal framework of Recipes stabilizes. As new functionality is needed in consuming projects like Admin or even third party providers for provisioning, we anticipate this is where folks will add it.

How providers and resources work

A provider is a service that contains resources. Services might be Gatsby, Contentful, the file system, or GitHub. A resource can be anything from a local file, to a Gatsby plugin, to a content model on a CMS.

Resources have a collection of methods that they export which Gatsby Recipes uses internally. They must implement CRUD.

- `create`: receives context and arguments which it uses to create a resource from scratch. When it successfully creates the resource, it returns the `read` with its new `id`.
- `read`: receives context and a unique identifier which it uses to fetch the resource.
- `update`: receives context, the `id`, and arguments. This function updates the resource, and when successful, returns `read` with the given `id`.
- `destroy`: receives context and an `id`. It first calls `read` for the resource in its existing state, and then removes it. It then returns the previously read object.
- `all`: optional method which returns an index of all resources.

In addition to CRUD, resources must also implement `schema`, `validate`, and `plan`.

- `schema`: Is a Joi object that specifies the shape of its properties.
- `validate`: Is a validation function that receives the potential properties before any CRUD function is called to ensure that they're valid.

- `plan`: Returns a diff of the resource from its current state and its desired state.

Each resource invocation adds its own diff to the plan. A plan is the composition of all resources used in a recipe. A resource will also have a `definition` which refers to the props or arguments that are passed to the resource.

When a plan is invoked, it will create or update all resources that were specified in the plan whether that's writing a file, updating a config, or provisioning something in the cloud.

Resource API signature

Below is the file resource edited for brevity to show the API signature for each of its methods.

The plan has an expected shape, and actions in CRUD (`create`, `update`, `destroy`) return the `read` for the resource after their actions. `destroy` is handled specially because we return the `read` value that occurs *before* the destroy happens so that you can leverage its previous state if needed.

As stated before, a resource can optionally include an `all` method which will retrieve an index and also result in an `allResourceName` field in the GraphQL API.

```
const create = async (context, { id, ...otherData }) => {
  /* ... */
  return await read(context, otherData.Path)
}

const update = async (context, resource) => {
  /* ... */
  return await read(context, resource.id)
}

const read = async (context, id) => {
  /* ... */
  return resource
}

const destroy = async context => {
  /* ... */
  return fileResourceBeforeDestroy
}

const all = () => {
  /* ... */
  return allTheResources
```

```
}

const plan = async (context, resource) => {
  /* ... */
  return {
    currentState,
    newState,
    describe,
    diff,
  }
}

const schema = {
  path: Joi.string(),
  content: Joi.string(),
  ...resourceSchema,
}

const validate = resource => {
  return Joi.validate(resource, schema, { abortEarly: false })
}

module.exports.plan = plan
module.exports.schema = schema
module.exports.validate = validate
module.exports.create = create
module.exports.update = update
module.exports.read = read
module.exports.destroy = destroy
module.exports.all = all
```

Resource list

See the recipes README →

Roadmap

The following features are next up for implementation.

Gatsby Admin GUI

Recipes included a WIP GUI that is currently being ported to Gatsby Admin where it is meant to live. This will (likely) be were most folks explore, read, and run Recipes.

See the PR →

Optimizing client bundles Part of the Recipes GUI inside Admin requires optimizing the client side bundle which includes sending along the precompiled JS from the server so that the client needs to include MDX and Babel transforms.

Inputs

Part of inputs are currently shipped, but we need to finish the implementation, especially adding the `useInput` API.

- `useInput` support for a more developer-friendly API
- CLI support so that inputs can be passed as CLI arguments

Statefile

The statefile will allow for Recipes to know which have been run and whether a resource needs to be updated or created. Right now, as implemented, Recipes will always call `create` on a resource because there's no way of deterministically knowing if it was created by a recipe or not.

It will need to track recipe history and recipe ids so that they can be tracked between runs, making a recipe with no changes run in an idempotent fashion.

Related

Alternatives considered

- **Code scaffolding** We looked into simple code generators as one potential design but we wished for a lot more power than they provided e.g. multi-step recipes and the ability to run recipes in both the client and browser.
- **Themes** were originally scoped to cover a lot of these ideas (a theme could specify everything needed for a section of a site and you could compose together themes) but we realized that most of what people would want to do when creating more complicated themes involved a lot of complicated one-off "setup" code that had no natural place with the standard Gatsby lifecycle. We needed something that sat outside of the normal Gatsby lifecycle.

Inspiration

- Terraform
- AWS CDK

Resources

This document aggregates a lot of different writing that @kylemathews and @johno have written regarding Recipes. Below are some of the original documents, and other relevant resources.

- Public Recipes RFC

- Gatsby Recipes README
- Gatsby providers and resources README
- WIP Engineering design doc
- Gatsby Recipes GitHub project

Glossary

Recipes has some terminology that we've defined below to help remove as much ambiguity as possible.

Definitions

Apply This is a mode modeled off of Terraform which applies the plan to the current environment. This is handled by the server when the client tells it to *"apply"*.

Plan Set of instructions, including the resource name, its definition, and the diff it will effect on the current state of the environment. For example, a plan for `<NPMPackage name="gatsby" />` will result in a plan that specifies *"NPMPackage"* as the resource name, a resource definition of *"package name gatsby"*, and the diff will be an ANSI-encoded git diff that shows the version change (if there is one).

This is the non-destructive version of apply which compares current state with intended state and reports back the *"diff"*.

Providers Providers refer to a service, whether that's something local to the development environment like the file system, or a third-party remote service like GitHub.

Providers contain a collection of resources.

Recipe An MDX file that contains instructions via components using a *Literate Programming* model.

Resources

Resources belong to a provider. Filesystem resources are files and directories. Gatsby resources include pages, plugins, and even site metadata.