All the code for Imgbot is available on GitHub. We will gladly accept contributions for the service, the website, and the documentation.

If you are unsure what to work on, but still want to contribute you can look for an [existing issue](https://github.com/dabutvin/Imgbot/issues) in the repo.

The following is where you can find out how to get set up to run locally as well as detailed information on exactly how Imgbot works.

### Imgbot Service

The core of Imgbot runs on a serverless stack called [Azure Functions](https://azure.microsoft.com/en-us/services/functions/).

The Function apps are running the image compression, pushing the commits, and opening the Pull Requests.

Once you get the tools you need to work with Azure functions you can run the apps locally.

You can either get the tools [integrated with Visual Studio](https://blogs.msdn.microsoft.com/webdev/2017/05/10/azure-function-tools-for-visual-studio-2017/) and use `F5`

or you can [get the CLI](https://github.com/Azure/azure-functions-cli) standalone and use `func run Imgbot.Function`.

If you are using Visual Studio for Mac there is [built-in support](https://docs.microsoft.com/en-us/visualstudio/mac/azure-functions) for Azure functions.

We also have support for running with VS Code. You will still need to get the CLI as mentioned above and the C# extension for VS Code in order to compile and get intellisense.

Each function has a task you can execute that will clean + build + run the process. To start one open the prompt with `cmd/ctrl + shift + p` and select `Run task`. From there you will see all the tasks checked into `.vscode/tasks.json`. Choose a function to run such as `Run CompressImagesFunction` and it will build and start up. To attach to this process choose the `Debug a function` configuration from the debug tab to see the running processes. Type `func` into the picker to see your running function and select it. It's a two-step process, the debugger and the function process. When you kill the debugger, the process will still be running. You can kill the function host by bringing up the prompt again with `cmd/ctrl + shift + p` and select `Kill the active terminal instance`.

Azure Functions operate on top of storage. To run the function locally you will need to bring your own storage account and add a `local.settings.json` in the root with `AzureWebJobsStorage` filled out and `FUNCTIONS\_WORKER\_RUNTIME` set to `dotnet`.

You can see the schema of this file in [the doc](https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local#local-settings-file)

Alternatively, running `func init` from the function directory will stamp out everything for you, or if you have a deployed function you can fetch the storage it is using.

`func azure functionapp fetch-app-settings <functionName>`

If you don't want to compile and run the `CompressImagesFunction` directly, you can use docker. See the image on [dockerhub](https://hub.docker.com/r/dabutvin/imgbot-compressimages).

There are a few additional environment settings that need to be set to run the compression workflow. These can be set with `local.settings.json` or any other way

- APP\_PRIVATE\_KEY - the secret certificate provided by GitHub during app registration

- PGP\_PRIVATE\_KEY - the secret key used for signing commits

- PGP\_PASSWORD - the secret password used for signing commits

- TMP - the location to do the cloning

Now that you are running the service locally, within the root of the repo you will see the following directories:

- `CompressImagesFunction` - The function that does the work of cloning, compressing, and pushing commits

- `Install` - The shared library to provide Installation Tokens to the functions

- `OpenPrFunction` - The function that opens Pull Requests

- `RouterFunction` - The orchestration layer

- `WebHook` - The function that is triggered from GitHub activity

The following file locations may be helpful if you are looking for specific functionality:

- `CommitMessage.cs` - generation of commit message compression report

- `CompressImages.cs` - clones the repo, compresses the images, pushes the commit

- `CommitSignature.cs` - uses a pgp private key and password to sign a commit message

- `ImageQuery.cs` - searches and extracts all the images out of the repo to be optimized

- `InstallationToken.cs` - uses a pem file and the GitHub API to get access to repos

- `LocalPath.cs` - generates the location to clone from

- `PullRequest.cs` - opens the pull request and sets the title and description

- `Schedule.cs` - logic to limit the frequency of Imgbot PRs

- `WebHookFunction.cs` - reads the GitHub hook messages and kicks off tasks

#### Triggers

Imgbot uses [QueueTriggers](https://github.com/Azure/azure-webjobs-sdk/wiki/Queues#trigger) to kick off workflows.

The triggers in place today are `routermessage`, `openprmessage`, `compressimagesmessage`.

Imgbot also uses an HttpTrigger as the entry point for the service.

#### Compression workflow

Imgbot uses [LibGit2Sharp](https://github.com/libgit2/libgit2sharp) to perform all the git operations.

The following is the high-level workflow for the `CompressImagesFunction`:

1. Clone the repo

2. Create the 'imgbot' branch

3. Run the optimize routine

4. Commit the changes

5. Push the 'imgbot' branch to the remote

The clone directory is read from environment variable `%TMP%` or falls back to `/private/tmp/` if this environment variable is not set.

Once the branch is pushed, Imgbot uses [Octokit](https://github.com/octokit/octokit.net) to create the pull request in GitHub.

#### Installation tokens

Imgbot uses [BouncyCastle](http://www.bouncycastle.org/csharp/) to help generate an [installation token](https://developer.github.com/apps/building-integrations/setting-up-and-registering-github-apps/about-authentication-options-for-github-apps/#authenticating-as-an-installation).

This requires a combination of a Private Key, an App Id, and an InstallationId to generate a token to be used on behalf of the installation.

The benefit of using installation tokens is that the user is in full control of the permissions at all times. We never have to store any tokens and they expire after about 10 minutes.

The installation token serves as a password to clone private repos, push to remotes, and open pull requests.

The username to accompany the installation token password is `x-access-token`.

For security reasons we cannot provide contributors with a pem file as this is a secret that delegates permissions in GitHub. You can run every part of the function except parts where authentication is required without this secret. If you are working on a part of the function that requires this secret then you can generate one for yourself to test with. [Register a GitHub app](https://github.com/settings/apps/new) for development purpose and install this app into the repo you are using to test with. Set the AppId in the code and you should be good to go.

If there is a part of this process that isn't clear or you have any questions at all feel free to open an issue in this repo and we'll work it out :)

#### Schedules

This Schedule class is responsible for throttling optimization routines.

Imgbot is triggered when there is a new image added to a repo and by default will submit a PR as soon as it can.

Some users prefer to defer the pull requests and do the optimization in bigger batches. This is implemented by offering three options 'daily', 'weekly', and 'monthly'.

Imgbot will check the commit log to find the last time we committed an optimization. If it has been long enough since Imgbot has last committed in this branch then we will try to optimize the images again. Otherwise we will skip running optimizations for this run.

#### Commit messages

Imgbot uses a standard commit title and generates a report of the image optimizations to be used in the commit message body.

The input is a dictionary where the key is the filename and the value is a pair of numbers that represent file size before and after compression.

This dictionary is transformed into an optimization report in the form of a commit message.

#### Image query

Imgbot locates all the images that are to be sent through the optimization routine with file directory access against a local clone.

The known image extensions are used to find all the images recursively and the ignored files from imgbotconfig are parsed.

#### Local Paths

For each execution, Imgbot generates the folder for the git operations to take place in.

Today this is done by combining the name of the repo with a random number.

#### Webhooks

The 2 main events we deal with through webhooks are installation events and push events.

Each time a repo has Imgbot installed, GitHub fires a hook to `WebHookFunction.cs` and we start the installation workflow.

Each time a repo that already has Imgbot installed gets pushed to, GitHub fires a hook to `WebHookFunction.cs` and, if the commit contains an image update, we start the compression worflow.

### Imgbot website

The frontend that drives https://imgbot.net/ is a generated static web app built with Grunt and a little bit of JavaScript. This static site is generated to be completely stand alone and hosted on a CDN for caching worldwide. The grid system for the Imgbot site is bootstrap 4. The purpose of this website is to run the landing page and docs for Imgbot.

You will find the `package.json` file for the website in the `Web/` directory of the repo. From here the input files live in the `src/` directory and the generated site is output to the `dist/` directory and git-ignored.

To kick off the generation

```

npm run gen

```

To start a lightweight dev server at `http://localhost:8888`

```

npm run serve

```

To compile the site on save

```

npm run watch

```

Within the `Web/` directory you will see the following key files

- `src/index.html` - the landing page markup

- `src/docs/layout.jst` - the docs page template

- `src/layout.jst` - the layout for the landing pages

- `src/css/site.less` - the landing page stylesheet

- `gruntfile.js` - the task configuration for generating and serving the site

### Imgbot Docs

The docs are published from checked in markdown files to the [Imgbot website](https://imgbot.net/docs) to view in a browser. Alternatively, the docs can be browsed and edited [within GitHub](https://github.com/dabutvin/Imgbot/tree/master/Docs).

When the docs are compiled to HTML they use the layout and metadata found in the `Web/src/docs` directory. The `metadata.json` file here will define the order and title of each doc. For example:

```

{

"slug": "contributing-website",

"title": "Contribute to Imgbot website"

}

```

The slug matches the name of the markdown file and also drives the URL segment to browse this doc.

This metadata file is read within the [Web/tasks/compile-docs.js](https://github.com/dabutvin/Imgbot/tree/master/Web/tasks/compile-docs.js) file. This is a custom grunt task that arranges all the documentation.

The template that renders the documentation is [Web/src/docs/layout.jst](https://github.com/dabutvin/Imgbot/tree/master/Web/src/docs/layout.jst). This is a template file that renders the documentation navigation and content as HTML.

### Tests

Imgbot uses VSTest with NSubstitue to manage unit testing of the logic within the Imgbot codebase.

The tests live in the Test directory.

Please do your best to add tests as new features or logic are added so we can keep Imgbot running smoothly.

:)