Mattermost

Developer Workflow

Developer Workflow at Mattermost Edit on GitHub

Common `make` commands for working with plugins

- `make dist` - Compile the plugin into a g-zipped file, ready to upload to a Mattermost server. The file is saved in the plugin repo's `dist` folder.
- `make deploy` - Compiles the plugin using the `make dist` command, then automatically deploys the plugin to the Mattermost server
- `make watch` - Uses webpack's watch feature to re-compile and deploy the webapp portion of your plugin on any change to the `webapp/src` folder
- `make test` - Runs the plugin's server tests and webapp tests
- `make check-style` - Runs linting checks on the plugin's server and webapp folders
- `make enable` - Enables the plugin on the Mattermost server
- `make disable` - Disables the plugin on the Mattermost server.
- `make reset` - Disables and re-enables the plugin on the Mattermost server.
- `make attach-headless` - Starts a `delve` process and attaches it to your running plugin.
- `make clean` - Force deletes the content of build-related files. Use when running into build issues.

You can run the development build of the plugin by setting the environment variable `MM_DEBUG=1`, or prefixing the variable at the beginning of the `make` command. For example, `MM_DEBUG=1 make deploy` will deploy the development build of the plugin to your server, allowing you to have a more fluid debugging experience. To use the production build of the plugin instead, unset the `MM_DEBUG` environment variable before running the `make` commands.

Developing in the plugin's webapp folder

`tsconfig.json`.

Exposing the Mattermost server using `ngrok`

When a plugin integrates with an external service, webhooks and/or authentication redirects are necessary, which requires your local server to be available on the web. In order for your Mattermost server to be available to process webhook requests, it needs to expose its port to an external address. A common way to do this is to use the command line tool ngrok. Follow these steps to set up `ngrok` with your server:

- Download the ngrok tool from here.
- Put the executable somewhere within your shell's `PATH`.
- With your Mattermost server already running, use the command `ngrok http 8065` to make your Mattermost server available for webhook requests.
- Visit the `https` URL from the `ngrok` command's output, and log into Mattermost.
- Set your Mattermost server's Site URL to the `https` address given from the `ngrok` command output.
- Monitor incoming webhook requests with ngrok's request inspector. Visit http://localhost:4040 once you have your tunnel open. You can analyze the contents of the HTTP request from the external service, and the response from your plugin.

If you're using a free ngrok account, the URL given by the output of the `ngrok http` command will be different each time you run the command. As a result, you'll need to adjust the webhook URL on Mattermost's side and the external service's side (e.g. GitHub) each time you run the command.

With this setup, many integrations require you to be logged into Mattermost using your ngrok URL. After logging into your ngrok URL pointed to your Mattermost server, in most cases you can continue using your `localhost` address in your browser for quicker network requests to your server. If you receive an error like `unauthorized` or `enable third-party cookies` when connecting to an external service, make sure you're logged into your ngrok URL in the same browser.

Using localhost.run instead of ngrok

If you would like to avoid using ngrok, there is another free option that you can run from your terminal, called `localhost.run`. Use this command to expose your server:

```
ssh -R 80:localhost:8065 ssh.localhost.run
```

An `http` URL pointing to your server should show in the terminal. The `https` version of this same URL should also work, which is what you will want to use for your webhook URLs. One

Debugging server-side plugins using `delve`

Using the `make attach-headless` command will allow you to use a debugger and step through the plugin's server code. A delve process will be created and attach to your plugin. You can then use an IDE or debug console to connect to the `delve` process. If you're using VSCode, you can use this `launch.json` configuration to connect.

```
{
    "name": "Attach remote",
    "type": "go",
    "request": "attach",
    "mode": "remote",
    "port": 2346,
    "host": "127.0.0.1",
    "apiVersion": 2
}
```

If the debugger is paused for more than 5 seconds, the RPC connection with the server times out. The server cannot communicate with the plugin anymore, so the plugin then needs to be restarted.

In order to be able to pause the debugger for more than 5 seconds, two modifications need to be done to the `mattermost-server` repository:

1. The plugin health check job needs to be disabled. This can be done by setting the server config setting `PluginSettings.EnableHealthCheck` to `false`. Note that if your plugin crashes, you'll need to restart it, using `make reset` for example. This command will also kill any currently running `delve` process. If you want to continue debugging with `delve`, you'll need to run `make attach-headless` again after restarting the plugin.

2. The `go-plugin`'s RPC client needs to be configured with a larger timeout duration. You can change the code at mattermost-server/vendor/github.com/hashicorp/rpc_client.go to increase the duration. Here's the change you can make to extend the timeout to 5 minutes:

```
sessionConfig := yamux.DefaultConfig()
sessionConfig.EnableKeepAlive = true
sessionConfig.ConnectionWriteTimeout = time.Minute * 5

mux, err := yamux.Client(conn, sessionConfig)
```