Plugin Development Patterns

This page lists common patterns you will use in your plugins very often, and are generally safe.

Sub-Observe

One of the most common patterns you'll see in shelter plugins is the pairing of a flux subscription and a DOM observation. This is because it makes it relatively easy to directly modify the UI.

The pattern looks like this:

```js
function handleElement(e) {
  // prevent handling the same element many times
  // :not in the selector below filters out in a performant way
  // if you're writing async code and you get duplicates, you can try add
  //     if (e.dataset.myPluginName) return;
  e.dataset.myPluginName = 1;

  const fiber = shelter.util.getFiber(e);
  // do whatever you want with the element here n stuff.
}

function handleDispatch(payload) {
  const unobs = shelter.observeDom("whatever:not([data-my-plugin-name])",
    unobs();
    handleElement(e);
  });
  setTimeout(unobs, 200);
}
```

```
const TRIGGERS = ["CHANNEL_SELECT", "LOAD_MESSAGES_SUCCESS"];
for (const t of TRIGGERS) shelter.plugin.scoped.flux.subscribe(t, handleD
```

This does a lot but is generally something you will see in a lot of places because it's a reliable way to modify elements on the page without a noticeable performance impact.

The fiber gives us a lot of useful context and information we can use to help fetch information we need.

If you've read the Your First Plugin guide, this should look quite familiar to you!

Some plugins with this pattern:

- Invidivizer, Yellowsink
- Channel Typing Indicators, ioj4

Dispatch Interception

Dispatch interception is a very efficient way to make client side changes involving modifying or preventing behaviours.

It tends to look something like this:

```js
shelter.plugin.scoped.flux.intercept(dispatch => {
  if (dispatch.type !== "CREATE_PENDING_REPLY") return;

  // modify `dispatch` directly or `return false` to block the dispatch.
});
```

The form this takes varies, be it modifying a detail of an action or intercepting the results of a data fetch.

Some plugins with this pattern:

- No Reply Mention, SpikeHD
- Timestamp File Names, ioj4
- Free Profile Colours, maisy

Store Patching

Often, you can achieve what you need by sitting in between a store and things that use it, by patching it.

This looks something like this:

```js
// give all users a discriminator based on their username
// running this example in your discord will make it crash and break in f
function process(user) {
  user.discriminator = user.username.length.toString().padStart(4, "0");
}

const unpatch1 = shelter.patcher.after(
  "getUser",
  UserStore.__proto__,
  (args, ret) => process(ret)
);
const unpatch2 = shelter.patcher.after(
  "getUsers",
  UserStore.__proto__,
  (args, ret) => {
    for (const id in ret)
      process(ret[id])
    }
);
```

Some plugins with this pattern:

- Prevent Spotify Pause, ioj4

HTTP Requests

shelter allows you to make and intercept HTTP requests before they hit the Discord API. This is very powerful and bridges the gap from client side changes via flux to server side changes via the API.

> **DANGER**
>
> Be careful! Making HTTP requests yourself has the potential to cause bans! Generally the risk of this is much lower via shelter than manually via fetch, but it's still there. Intercepting requests should be low risk, but it is suggested to attempt to test an action yourself for real, and attempt to mimic that. Do not use Discord's developer docs as a resource as some of the endpoints are bot-only, and are different for users.

> **WARNING**
>
> These are specifically for Discord endpoints only, not for anything else. Do not try to use these for general fetching.

You can make a request yourself, such as muting a guild:

```js
shelter.http.patch({
  body: { muted: true, suppress_everyone: true, suppress_roles: true },
  url: `/users/@me/guilds/${guild_id}/settings`
})
```

You can also modify an outgoing HTTP request, such as modifying messages:

```js
// makes you more happy :)
shelter.http.intercept("POST", /\/channels\/\d+\/messages/, async (req, s
  req.body.content = req.body.content.replaceAll(":(", ":)");
  const response = await send(req);
  return response;
});
```

Or even just block requests entirely:

```js
shelter.http.intercept("POST", "/science", () => {});
```

Some plugins with this pattern:

- Antitrack, Yellowsink
- Mute New Guilds, taskylizard
- Text Replacements, Yellowsink

Long Lived Observers

The DOM Observer is intended to be used momentarily to listen for an anticipated change to the page. Some plugins skip the step of waiting for reason to believe a change will happen, and just add a listener at plugin load that lives until unload.

Basically, its the same as sub-observe, but without the flux part:

```js
function handleElement(e) {
  e.dataset.myPluginName = 1;

  const fiber = shelter.util.getFiber(e);
  // do whatever you want with the element here n stuff.
}

shelter.plugin.scoped.observeDom("whatever:not([data-my-plugin-name])", h
```

> **WARNING**
>
> This pattern simplifies the development process and increases reliability, *however* it means shelter has to watch the page constantly for as long as your plugin is running, which could in theory decrease performance. In my testing (on a reasonably modern system, in Firefox) it has an impact too small to measure in practice.

Some plugins with this pattern:

- Open Profile Images, ioj4
- Freemoji, Yellowsink
- GPT, edde746

Cleanup Reinsertion

To have your component persist throughout rerenders of it's parent element you can reinsert your component when solid's `onCleanup` gets called.

To make `onCleanup` function properly, you'll need to wrap your component in shelter's `ReactiveRoot` block.

The pattern can look like this:

js

```
function MyComponent() {
  // queueMicrotask to give React enough time to finish the rerender
  shelter.solid.onCleanup(() => queueMicrotask(insertComponent))
  return (
    <ReactiveRoot>
      <h1>My Header</h1>
    </ReactiveRoot>
  )
}

function insertComponent() {
  // see warning below
  if (!isPluginEnabled) return;

  const parent = document.querySelector(`[class*="whatever"]`)
  if (parent) {
    parent.append(<MyComponent />)
  } else {
    // long lived observer here just in case
  }
}
```

WARNING

When using recursive methods in your plugins, make sure to check whether your instance of the plugin is still enabled otherwise it will continue to run after disabling the plugin.

Some plugins with this pattern:

- VC Timer, ioj4

Previous page
Settings, Storage & UI

Next page
Ideals