Your First Plugin

This guide will walk through writing a simple but functional plugin from start to finish. It assumes you have shelter installed, Lune setup, and your monorepo setup.

Orientation

Assuming you have a fresh template plugin repo setup, you'll have a folder `plugins/hello-world` . Inside here is the scaffold for an empty shelter plugin.

The `plugin.json` file contains all the user-facing metadata for your plugin: the name, author, description, etc. This isn't code but is still important.

The `index.jsx` file is the entry-point of your plugin, the main file. This file can be renamed to have any extension of `js` , `jsx` , `ts` , or `tsx` , to your tastes.

Notice the three main things in this file: a destructure of the shelter global, this is where you'll be getting your APIs from, the `onLoad` function, which is ran when your plugin is enabled, after all the top level code finishes, and the `onUnload` function, which is ran when your plugin unloads.

You can also add a `settings` export, to add a GUI settings menu to your plugin. All of these exports are optional, shelter will use what you provide but will not complain if you don't provide them.

Let's get writing then

We're going to build `show-username` from scratch, which displays the username of people with nicknames in chat.

So, `lune init show-username` and open it in your text editor of choice.

The way we'll approach this is the same as most shelter plugins:

- listen for a reason to suspect the document is about to change: Flux
- wait for the document to change: the DOM Observer
- pull helpful information out of *react fibers*
- modify the document manually

TIP

You can read more about common patterns here.

So let's import the APIs we're going to need, which isn't particularly interesting on its own, just object destructures:

```js
const {
  GuildMemberStore,
  ChannelStore,
  SelectedChannelStore,
  RelationshipStore
} = shelter.flux.storesFlat;
const {
  util: { getFiber, reactFiberWalker },
  observeDom
} = shelter;
const { subscribe } = shelter.plugin.scoped.flux;
```

The flux stores let us easily query information from Discord, that is helpful to us. The fiber utils are what we'll use to pull more info off of the page. The observer lets us wait until the document is ready to be modified, but also run immediately when it is so that users see no flash / jank.

The scoped subscribe function lets us listen for flux dispatches, and automatically cleans up on unload for us.

Listening for Flux dispatches

We're going to listen for a list of flux event types that are of interest to us, which signal that the document *might* be about to change.

When something needs to change the page, React, the UI Framework Discord uses, does something called a rerender. This will wipe all our modifications off the page in the majority of cases.

Luckily, almost all of these rerenders come following a flux dispatch, so we can listen for these same dispatches too, and then swoop in after React is done to apply our changes.

We'll start with just our simple listener:

```js
function handleDispatch(payload) {
}

const triggers = ["MESSAGE_CREATE", "CHANNEL_SELECT", "LOAD_MESSAGES_
for (const t of triggers)
  // the subscribe function we imported earlier
  subscribe(t, handleDispatch);
```

Before we work too hard on implementing our logic, we want to refine our subscriptions. In particular, `MESSAGE_CREATE` is very noisy, so we'll ignore any that aren't for our current channel.

```js
function handleDispatch(payload) {
  // only listen for message_create in the current channel
  if (payload.type === "MESSAGE_CREATE" &&
    payload.channelId !== SelectedChannelStore.getChannelId()
  )
    return;

  console.log("We might need to do some work here!");
}
```

Now we have the ability to know when the change is likely to happen shortly, but we don't know at what point it has definitely happened, how much changed, what changed, or where it changed, and that it won't be changed further yet.

For example, `UPDATE_CHANNEL_DIMENSIONS` generally just means the window has been scrolled or resized, which in many cases means we have new messages displaying, but not necessarily!

DOM Observation

Now we need to wait for the page to actually be updated. This can be done using a few different methods, but the one in shelter is the `observeDom` API. You provide it with a CSS selector, and it will run a callback you pass for every element which either changes, or is the child of an element that changes.

```js
function handleElement(elem) { }

function handleDispatch(payload) {
  // only listen for message_create in the current channel
  if (payload.type === "MESSAGE_CREATE" &&
    payload.channelId !== SelectedChannelStore.getChannelId()
  )
    return;

  const unObserve = observeDom("[id^=message-username-]", (elem) => {
    handleElement(elem);
    unObserve();
  });

  setTimeout(unObserve, 500);
}
```

So, a lot more is going on here than it might look like. We start watching the document for any element that changes that has an ID starting with `message-username-` . Document changes *come in batches*, and when the next batch of changes come in which a matching element is found, we pass every match to the callback you pass. You only get one element at once.

We pass the element off to a function to do later work, fine, but the behaviour of this `unObserve` function is notable. You may assume that once its called here, that's it, and your callback will never get called. Actually, this isn't quite true - it makes plugins a lot more comfy to write generally if this function will only stop AFTER the current batch has been processed.

So we can call `unObserve` inside our callback to clean the observer up as soon as its finished doing its job, but still getting all of the element we want, not just one.

Finally, we use a `setTimeout`, because we are assuming that the change we want is going to happen within half a second, and if a change *isn't* going to happen, we don't want to leave loads of observations dangling for performance reasons:

- It can cause way lots of duplicate work
- It slowly builds up callbacks over time so performance slowly decreases and memory usage increases.
- It forces shelter to watch the document 24/7, which is slow, instead of only bothering watching when requested to

So now, we have access to every message element *immediately* after React updates it on the page.

Pulling data off of the page

So, we need to know what user we're working with before we can go about inserting their nickname onto the page.

If you want to read more about fibers, I suggest reading the background guide, but to TL;DR the relevant part, Discord are using React, and that means there are *three* trees of elements at work that make up the page:

- The *DOM*, or just document, which is the raw JavaScript interface to the page, which is what we are working with mostly in shelter, exposed on `window.document`. This reflects precisely what the user is actually seeing.

- *React elements* (a.k.a "The Virtual DOM"), which is the tree that Discord's code builds directly via JSX syntax. This exists because modifying it is much cheaper than modifying the real DOM. (The extent to which this is advantageous is arguable, but it allows React to throw away and reconstruct the entire tree every time (lol)).
- *React Fibers*, which are much more obscure, and are internally used by React to speed up the Virtual DOM. They basically contain all of the information of the elements, but with extra internal info, including:
  - the type of the element (be it nothing, a React built in element, DOM element, or React component)
  - the props passed to that element (the same kind of objects you'll find in Flux stores)
  - the corresponding DOM element, if there is one
  - the parent, child, and siblings
  - etc.

The elements tree is very powerful to modify, and is in fact how client mods used to do their work before September 2022, (and how some - less sustainable - client mods still do!). Unfortunately, it is volatile and, since the death of webpack searching, difficult.

If you need any convincing, webpack searching came *back again* recently (late October 2023), and every plugin and mod that used webpack searching (or regex patching, an alternate means to get to element tree patching) broke instantly due to the change. shelter didn't. 😉

The document is the easiest and safest thing to modify, but contains missing information. It's purely what the user needs to see, and in this case, that is not the username of the user!

We cannot really modify the fiber tree ourself (or rather, nobody has figured out how to do it yet...), but it does contain information from the element tree that can really help us. We can also easily get onto it from the document, so it's really good for getting extra context once we have a document element.

This includes things like full message, channel, and guild objects, functions we could try calling, etc etc.

Let's temporarily leave behind the world of elements and dive into the (highly detailed) fiber tree:

```js
function handleElement(elem) {
  const fiber = getFiber(elem);
}
```

And we want to dig for the author ID, channel ID (so we can find the guild ID, to check the server nickname), and the message type (DM or channel).

If you look at this fiber, you'll notice we don't have any of this in the fiber! What gives!

Well, the fiber we're currently holding corresponds directly to a DOM element, but we actually want the one that corresponds to a React component, which has the useful info. We need a different, but related fiber.

See:

```jsx
function MyReactComponent(props) { // -- and we want the fiber right at t
  // stuff in here
  return (
    <AnotherComponent> // -- but this is also a fiber, and who knows how
      <div> // -- we might be holding this fiber, which corresponds to th
        hi!
      </div>
    </AnotherComponent>
  );
}
```

So we need to walk up the tree until we find what we want.

shelter provides the fiber walker to make moving up and down the tree much easier. We will move *up* the tree to find a fiber with a prop named `message` , and get that object:

```js
function handleElement(elem) {
  const fiber = getFiber(elem);
  //                                /- fiber        /- filter  /- go 
  const message = reactFiberWalker(getFiber(elem), "message", true)?.
  if (!message) return;
}
```

If message is undefined, we've somehow missed, and walked up so far we've hit the very very top of the tree. We'll just check to be sure.

Context from Flux stores

The message object is pretty big, but the bits of it we care about look like this:

```json5
{
  author: { id, username },
  channel_id
}
```

We are going to look up the channel type and guild ID by asking the Flux store for info, and the nickname. This comes direct out of the same sources of truth Discord itself uses.

```js
const { type, guild_id } = ChannelStore.getChannel(message.channel_id
// type = 0 -> guild, type = 1 -> DM
const nick = type
  ? RelationshipStore.getNickname(message.author_id)
  : GuildMemberStore.getNick(guild_id, message.author_id);

if (!nick) return;
```

If the user has no nickname, we know now we can stop. The user's real username is already on display.

Changing the page

Now, we can insert into the UI - note we're using Solid JSX here, so we're working with real document elements, not React elements or anything:

jsx
```jsx
elem.firstElementChild.textContent += ` (${message.author.username})`
```

Mutexing

You may have noticed that you get duplicates of our changes appear. This is because we may detect a change that doesn't actually wipe our modifications. To handle this, we place something on the element that, if React were to reset our changes, it would also remove. This is usually a dataset attribute.

This is a side effect of how we are doing things - we don't intercept the UI rendering, we modify after it. This inherently means we are out of sync with React. This can cause many issues, and this is one of them.

So right at the very start of our handleElement call, we add a check, and add an element:

js
```js
function handleElement(elem) {
  if (elem.dataset.showuname) return;
  elem.dataset.showuname = 1;

  // ...rest of function
```

And let's put it all together!:

js
```js
const {
  GuildMemberStore,
  ChannelStore,
  SelectedChannelStore,

  RelationshipStore
} = shelter.flux.storesFlat;
const {
  util: { getFiber, reactFiberWalker },
  observeDom
} = shelter;
const { subscribe } = shelter.plugin.scoped.flux;

function handleElement(elem) {
  if (elem.dataset.showuname) return;
  elem.dataset.showuname = 1;

  const fiber = getFiber(elem);
  const message = reactFiberWalker(getFiber(elem), "message", true)?.
  if (!message) return;

  const { type, guild_id } = ChannelStore.getChannel(message.channel_
  // type = 0 -> guild, type = 1 -> DM
  const nick = type
    ? RelationshipStore.getNickname(message.author_id)
    : GuildMemberStore.getNick(guild_id, message.author_id);

  if (!nick) return;

  elem.firstElementChild.textContent += ` (${message.author.username}
}

function handleDispatch(payload) {
    // only listen for message_create in the current channel
    if (payload.type === "MESSAGE_CREATE" &&
            payload.channelId !== SelectedChannelStore.getChannelId()
    )
        return;

    const unObserve = observeDom("[id^=message-username-]", (elem) =>
        handleElement(elem);
        unObserve();
    });

    setTimeout(unObserve, 500);
}
```

```
const triggers = ["MESSAGE_CREATE", "CHANNEL_SELECT", "LOAD_MESSAGES_
for (const t of triggers)
  subscribe(t, handleDispatch);
```

You'll notice, this is about the same as the actual show-username plugin!

Finishing up

This is a basic plugin, but it shows you a common pattern used to build shelter plugins, and gives you a taste of the general approach used. I suggest heavily you keep reading the other guides, which will help your general understanding of shelter development.

Feel free to have a look at what plugin devs are doing in their plugins, and try your hand at bigger things.

Another thing you may wanna play with is that you have all of Solid at your fingertips to build reactive apps easily, or even just to replace the annoying process of creating DOM structures by hand with `document.createElement` with something like the following:

```jsx
someElement.appendChild(
  <div style={{ margin: "5rem" }}>
    <span>Hi guys</span>
    <button onClick={() => console.log("ping!")}>click here!</button>
  </div>
);
```

And because Solid is reactive, if you use a signal in that UI, then updating the signal will automatically update your injected UI on the document, both making building reactive UIs easy, and making, injecting an element *now* and adding content *later* really easy!

Oh, and if you're doing anything complicated with Solid, and just manually placing it onto the document (not passing it to shelter settings or modals), then you should probably use a shelter UI `<ReactiveRoot>` ! 😉

But enough about Solid, go forth and improve Discord, and I hope you enjoy developing for our mod!

-- Yellowsink

Previous page
Lune

Next page
Settings, Storage & UI