# The problems you will have when creating a plugins system for your shiny UI project

Joaquim Rocha

Principal SWE Manager at Microsoft

floss.social/@jrocha

## Here to share, not to solve.

Goals:

- Identify patterns to help you get ahead of likely outcomes when creating a plugins system

Non-goals:

- Show you how to create a perfect & secure plugin system

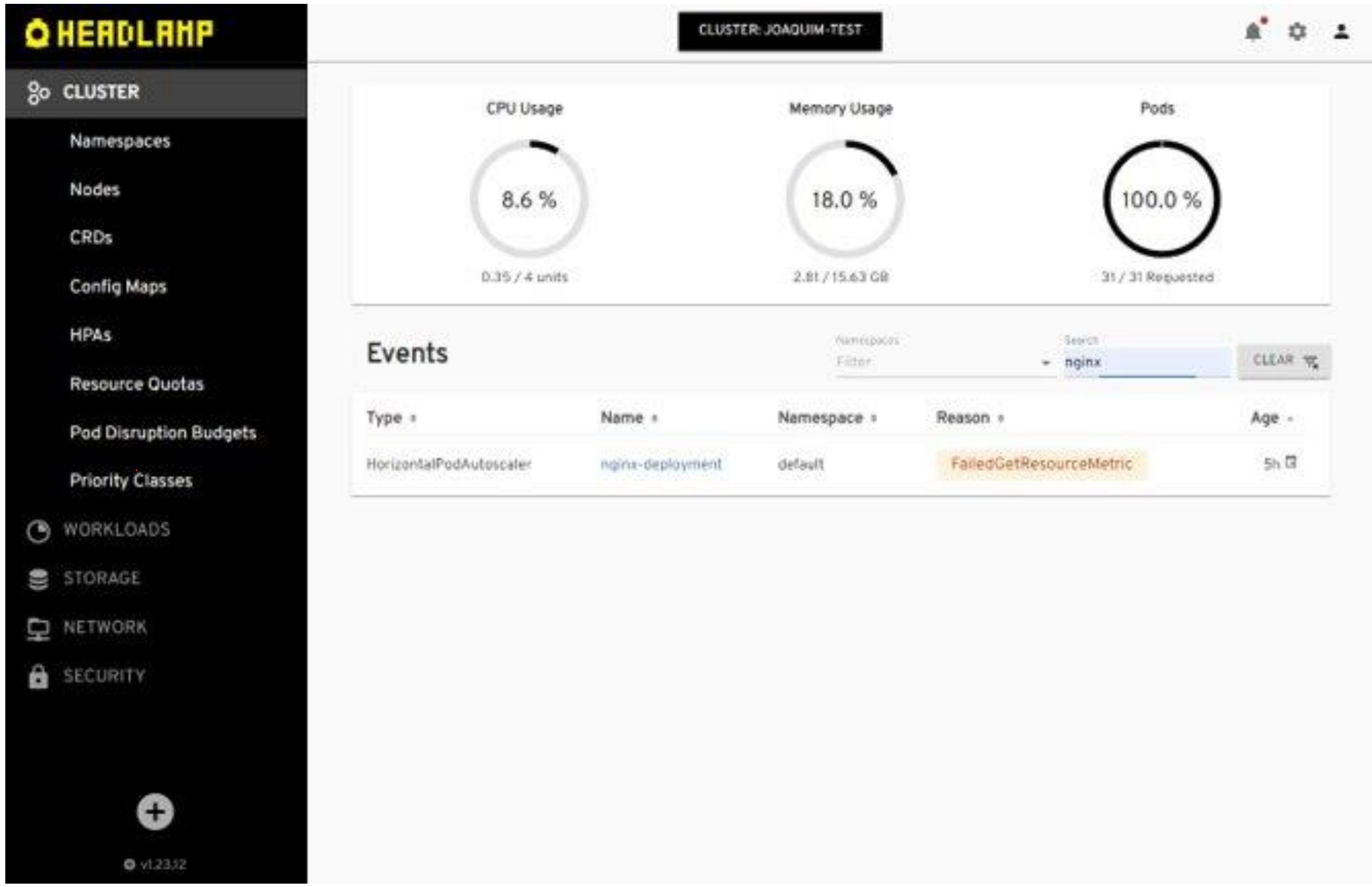
# Setting up the context

There are many systems using plugins in JS: VSCode, Mattermost, ...

**Us:**

Headlamp is an extensible Kubernetes UI

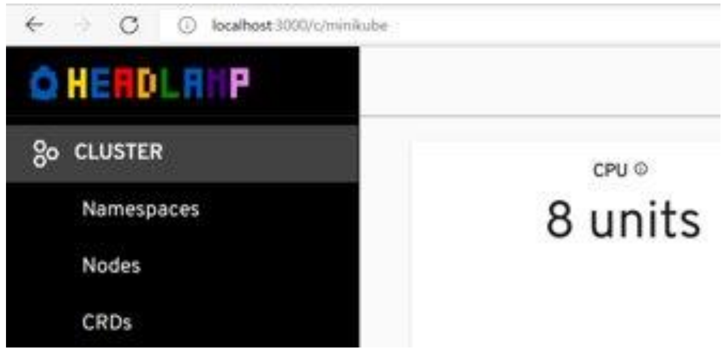
- Has a backend (go) and a frontend (Ts/React).
- Can be run as a desktop app (Linux, Mac, Windows)
  Or deployed as a web app
- [headlamp.dev](https://headlamp.dev)



Events

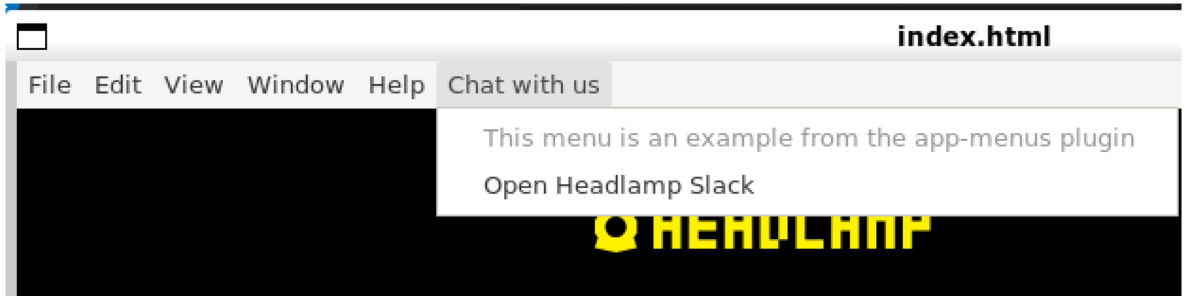
| Type | Name | Namespace | Reason | Age |
|---|---|---|---|---|
| HorizontalPodAutoscaler | nginx-deployment | default | FailedGetResourceMetric | 5h |

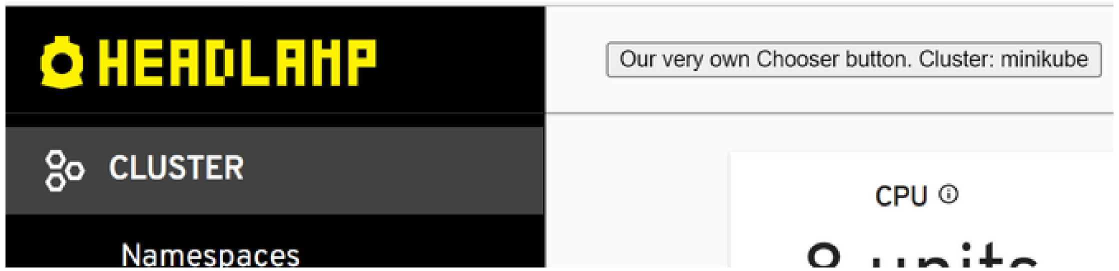
# Setting up the context: What do we mean by plugins

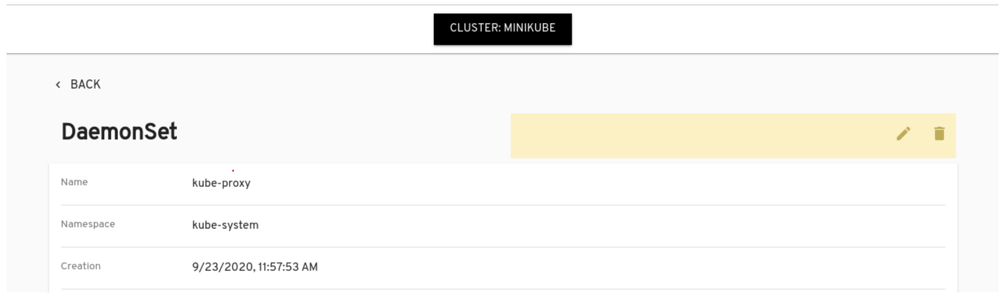


Plugins should:

- Be loaded dynamically
- Change the functionality through an API
- Can change the UI or other core functionality







| | |
|---|---|
| Name | kube-proxy |
| Namespace | kube-system |
| Creation | 9/23/2020, 11:57:53 AM |

## PLUGIN ANATOMY

## We need the code, what about info? (captain obvious warning!)

- The code: bundled single JS file
    - Ready to be run
    - Already includes any needed dependencies
- The info/manifest: *package.json*
    - Already has the base info needed in most cases
    - Do not duplicate the info by requiring info declaration as part of the plugin code
    - Being a separate, textual file, means we can read it without having to evaluate the plugin's code (avoid having any info/metadata coming from the code)

## LOADING / UNLOADING PLUGINS

## Loading a plugin

- Should the code just "run free", or be required to have an *activate* method?
- **With** an *activate* method
  - Tells the plugin developer exactly when the main plugin code is to be executed
  - May tell the system if the plugin was successfully loaded
    - By having the activate method return a state, for example
- **Without** an *activate*:
  - Loading the code itself is the activation!

## Deactivating

- What about *deactivate*?
  - Should allow the plugin to stop any ongoing work
  - Can be used as a clean-up method
  - Likely unused by most plugins

OTOH, **Deactivating != Unloading**

- Unloading means returning to the state before the plugin was loaded
  - This is highly a responsibility of the system
  - May involve reloading without said plugin

## Conclusion: Loading & Unloading a plugin

- *activate*/*deactivate* are highly about giving control to the developer, not the system
- The system should assume that code gets loaded anywhere and anytime
  - and that it doesn't get deactivated properly by itself

## API / PLUGIN STRUCTURE

# Object-oriented or Functional?

- A Plugin class sounds like a reasonable idea
- But the world is going functional? (Ultimately is a taste matter)

```
class Plugin {
  activate(registry: PluginRegistry) {
    if (new Date().getDate() !== 1) {
      return [false, 'Our plugin only works
on Mondays!'];
    }
    const SnoozeButton = () => ...
    registerHeaderAction(SnoozeButton);

    return [true, 'All good'];
  }
}

registerPlugin(Plugin);
```

```
export function activate(registry: PluginRegistry)
{
  if (new Date().getDate() !== 1) {
    return [false, 'Our plugin only works on
Mondays!'];
  }
  const SnoozeButton = () => ...
  registerHeaderAction(SnoozeButton);

  return [true, 'All good'];
}
```

## What if plugins are an actual React component?

- Built-in lifecycle: can be used to implement *activate*/*deactivate*
- Use of hooks directly in the actual plugin itself

```
export const MyPlugin = () => {
  useSomeOtherHook();

  useActivate(() => {
    if (new Date().getDate() !== 1) {
      return [false, 'Our plugin only works on Mondays!'];
    }

    const SnoozeButton = () => ...
    registerHeaderAction(SnoozeButton);

    return [true, 'All good'];
  });
};
```

## Declarative or Imperative?

```
class Plugin {
  topBarActions = [
    {
      label: 'Delete',
      icon: 'delete-circle',
      type: Actions.Types.Button,
      action: Resource.delete,
      actionArgs: [Resource.getID]
    },
  ];
}
```

```
const button = () => (
  <Button
    label="Delete"
    icon="delete-circle"
    onClick={() => {
      Resource.delete(resource.id);
    }}
  />
);

registerTopBarAction(button);
```

- Declarative approach: may make plugins simple to learn but require more maintenance
- Imperative approach: offers more flexibility but arguably less control by the system

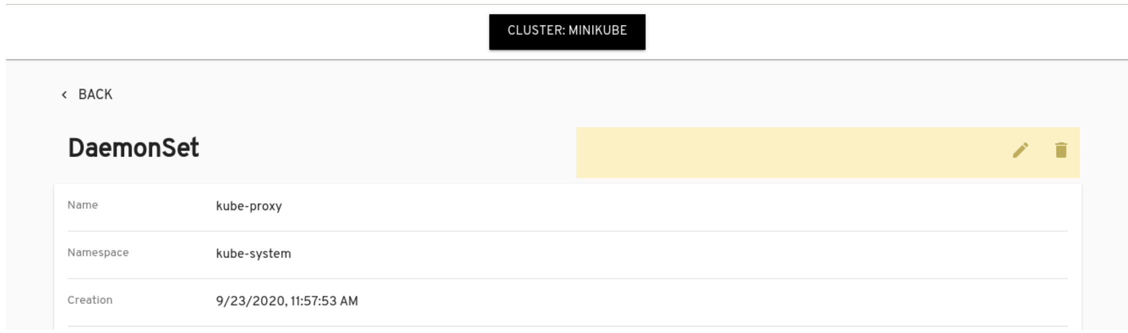
## API / FUNCTIONALITY

## API for plugin functionality

- Think about all the operations plugin devs may need
- Likely they will end up needing all counterparts to every op you offer
    - i.e. if you allow to add header actions, there will likely be a need for removing or updating them too.
    - Some sort of CRUD...
- What should the API look like though?

# Example: You support a list of header actions

- Should you have one function per operation?
- The following are the creation actions:



| | |
|---|---|
| Name | kube-proxy |
| Namespace | kube-system |
| Creation | 9/23/2020, 11:57:53 AM |

```
const button = () => (
  <Button
    label="Delete"
    icon="delete-circle"
    onClick={() => {
      Resource.delete(resource.id);
    }}
  />
);

registerHeaderAction(button);
```

Or maybe:

```
registerHeaderActions([button1,
button2, button3]);
```

## Example: Removing a header actions

- What should the deletion actions be?

Maybe?

```
deregisterHeaderAction(button);
removeHeaderAction(button);
```

- However, can a plugin easily identify any actions not added by itself?
  - Relying on a function's name may not work (when the code gets minimized)
  - **Solution:** Add IDs to any functionality you may need to refer to.

Like:

```
registerHeaderAction({id: 'my-delete', action: button});
```

## CRUD(S?) (CRUD + whatabout Shuffling)

**Random 1st time plugin developer on the internet:**
*"Hey there. Great program. How can I add my header item as the 1st one instead of being appended at the end?"*

## Example (cont): You support a list of header actions

- Don't add an index parameter to the functions...
- **Possible solution:** A "list processor" instead of a function for every op

```
const MyDeleteButton = () => (
  <Button
    label="Delete"
    icon="delete-circle"
    onClick={() => {
      window.alert('Not today!');
    }}
  />
);
```

```
const changeDelete = (actions: HeaderAction[]) => {
  return [
    {
      id: 'my-delete',
      action: MyDeleteButton,
    },
    ...actions.filter(action => action.id !== 'delete')
  ];
};

registerHeaderActionsProcessor(changeDelete);
```

## DEVELOPER EXPERIENCE

## Developer Experience

- Providing a plugin manager program is a good idea
  - This can help start plugins but also check compatibility, etc.
  - Headlamp ships *@kinvolk/headlamp-plugin*
  - This allows to create, update, and run a plugin.

- Require developers to configure as little as possible, especially infrastructure
  - The less the system requires/allows to be configured, the more control the system has
  - Results in a better dev exp and less breakage

# Developer Experience

- Don't just generate the boiler plate, avoid it!
- Ship any default, not-likely-to-be-changed, files in your dev dependency (and point to them)

**package.json:**

```
{
  "name": "change-logo",
  "version": "0.0.1",
  "description": "Changing the logo in Headlamp can be done like 
this.",
  "scripts": {
    "start": "headlamp-plugin start",
    "build": "headlamp-plugin build",
    "format": "headlamp-plugin format",
    ...
  },
  "prettier": "@kinvolk/eslint-config/prettier-config",
  "eslintConfig": {
    "extends": [
      "@kinvolk",
      "prettier",
      "plugin:jsx-a11y/recommended"
    ]
  },
  "devDependencies": {
    "@kinvolk/headlamp-plugin": "^0.5.4"
  }
}
```

**tsconfig.json:**

```
{
  "extends": "./node_modules/@kinvolk/headlamp-
plugin/config/plugins-tsconfig.json",
  "include": ["./src/**/*"]
}
```

## BUILDING & BUNDLING JS

## Bundling JS

- Bundling JS is easy with webpack *(kind of)*!
- But plugins will run within your app
  - You don't want them to bundle any modules your app has
  - This means its own lib and dependencies (React, react-router, redux, material-ui, …)

# Avoid bundling everything

- Headlamp uses webpack's *external-modules* to indicate where to find dependencies:
  - E.g. mapping react-router-dom to window.pluginLib.ReactRouter

- Also avoided shipping our entire Headlamp modules in the plugin's lib NPM package: shipped just the type declarations...
  - This made testing plugins very difficult: cannot be tested directly as their dependencies are not available to compile it
  - **Possible solution:** Just ship the lib and use it as an external module + add the infra for testing the plugin directly.

## RUNNING THE PLUGINS

## Compatibility

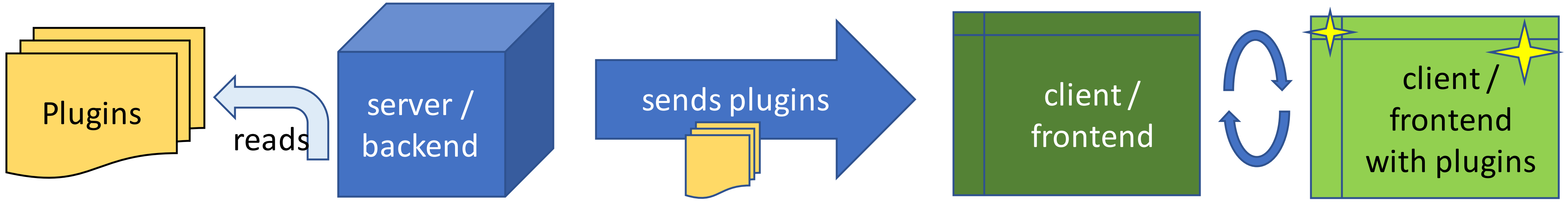
- Once beyond the 0.X versions, make sure compatibility is verified before loading plugins (or else...!)
- Add it to *engines* in the *package.json*
  - So you can check the compatibility before loading any code

**package.json**

```
{
  "name": "my-plugin",
  ...
  "engines": {
    "my-plugin-system": "^1.5"
  },
  ...
}
```

## How to run the system + plugins

- Highly special to each project
- Here is how Headlamp does this:



# THANK YOU!

headlamp.dev