Aurox Wallet Extension Architecture Overview

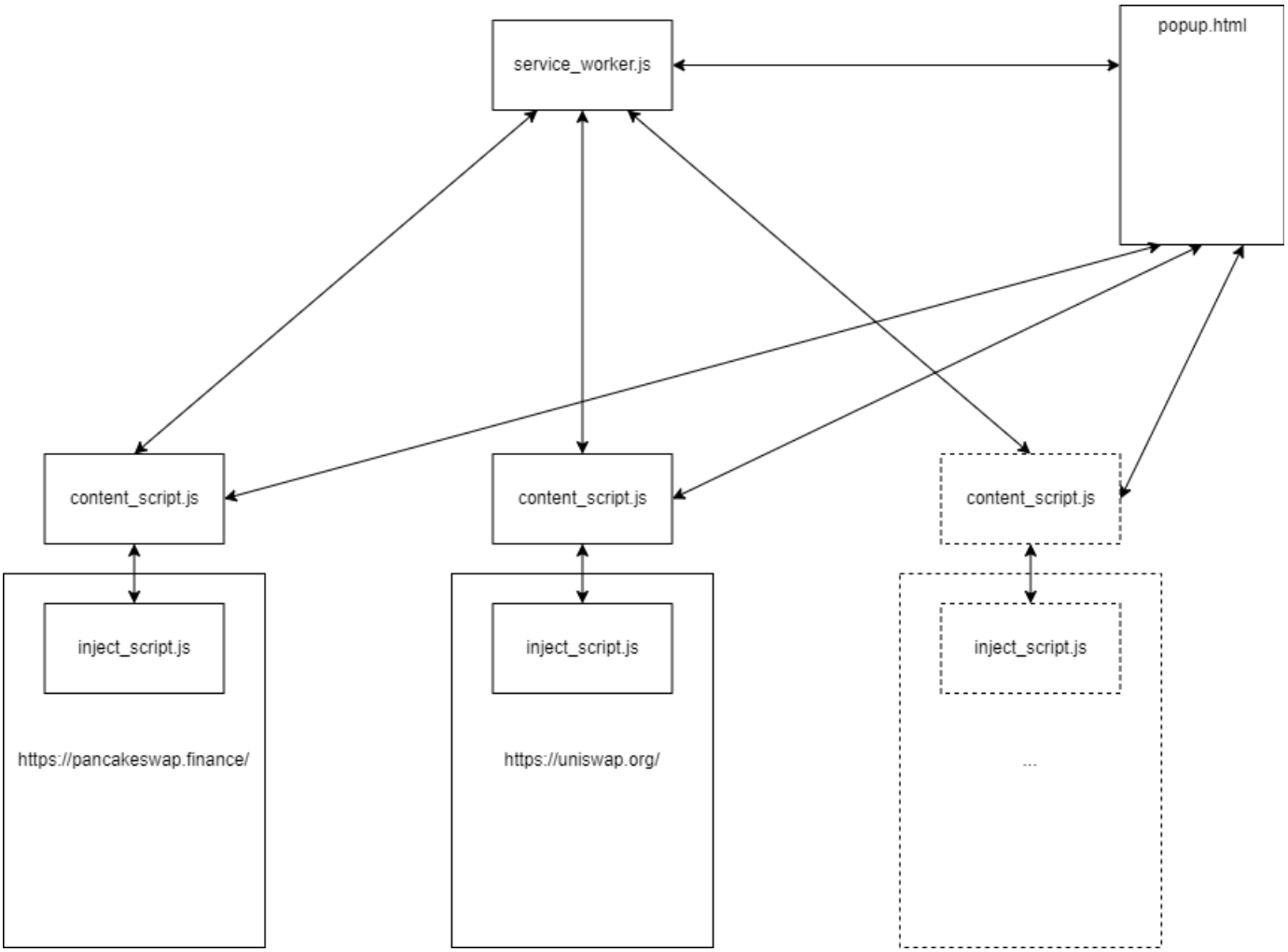
General Overview

To understand why some design decisions have been made, we must first explore the way browser extensions work in the first place. As defined by the extension manifest v3, we can have the following entities within an extension deployment:

1. **default_popup**: This is a web-view that gets opened as a panel window by the chrome when the user clicks on the extension **action** (the button to the right side of the search box when the extension is pinned). It's **a simple webpage with an html file associated with it**, which comes from a path within the extension deployment bundle. Every time the user clicks on the action that web-view is constructed and rendered like a normal webpage. Once the user clicks away, it's completely destroyed along with all it's internal state. Therefore, in order to keep that state across clicks and clicks-away we must do extra work.
2. **service_worker**: This refers to a script that is run for the extension as it's background worker. It also acks as a normal service-worker, there is only ever one instance per chrome executable (not per tab, nor per window). This means, **a single service-worker instance is shared across all entities** of the extension. There is however an ongoing issue with it where it is forcefully stopped after 5 minutes and must be manually resurrected, which complicates a lot of things.
3. **content_script**: This is a script that runs in an isolated context with access to the dom of a webpage (chrome tab or a frame within a tab) that is run per each page that matches the criteria defined by the extension manifest, in our case any https website. This script **has access to the browser extension APIs** and it also **has access to the DOM of the target website but NOT to it's execution context.** Therefore, in a sense it's an intermediator between the untrusted context of the website and the trusted context of the extension. However, it's not fully trusted either. Anything originating from content_script must be treated as a potential attack.
4. **inject_script**: This script gets injected into the context of the webpage. It **has complete access to the browser context and the browser context has complete access to it**. Therefore, it can be used as an API for the website to communicate with the extension, for us this acks as the web3 provider. It goes without saying that this script is synonymous with the website's script, as trustable as the website itself. The only way this can be executed in the browser context is to have content_script add a script tag to the website which references the file within the extension bundle. manifest must explicitly allow this.
5. other entities: The extension can have any number html pages and content_scripts that can be accessed/opened according to some js action such as opening a window or tab, or some declarative manifest definitions to inject the file into a website as content scripts.

Each of these entities run in their own isolated context, they do have access to some browser extension APIs, but they can't directly access each others memory space. For this reason, we will need a communication channel based on what can connect to what.

An overview of how these entities come together:



Note: The communication between content_script and inject_script is via DOM events, all the others are via browser extension APIs.

The folder structure for the entities are organized as follows:

```
src/
  /common  ......  Code shared between all the entities (communication,
utils, etc..)
  /content_script
  /inject_script
  /service_worker
  /ui
    /common  ...  Common code shared between frame entities (anything
with UI)
    /...
    /frames
      /popup
      /connect
      /hardware
```

Bundling

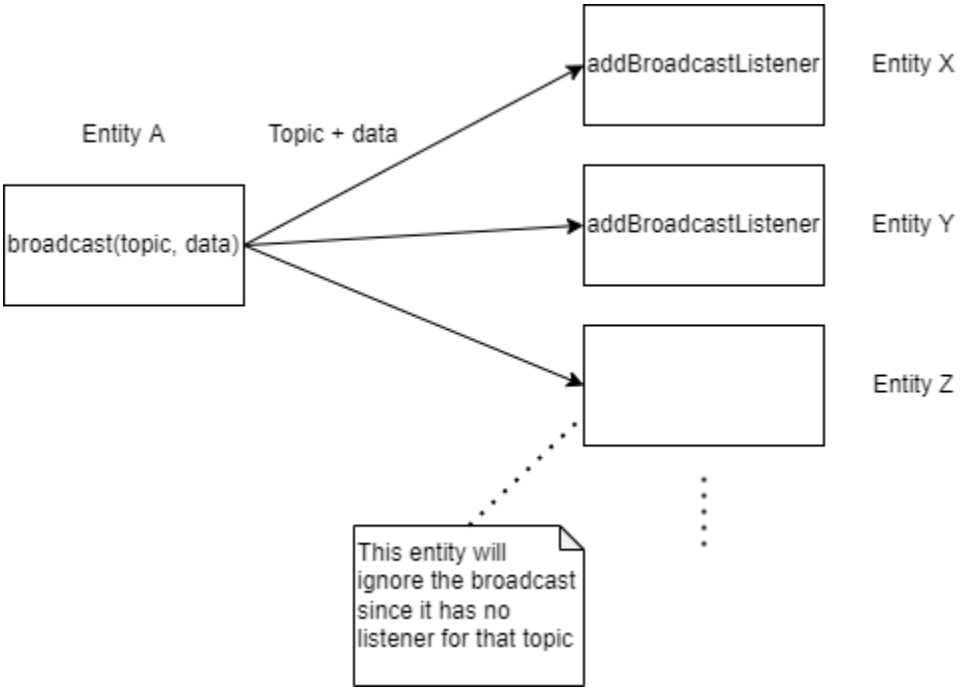
These entities are bundled by webpack. It's worth keeping in mind that script (content_script, inject_script, service_worker) entities shouldn't be broken down into smaller chunks as that will require automatic manifest generation as well. which is not desirable since that will cause each update to have a lot changes to the manifest file. The only entities that may utilize chunking are the frame entities (popup, hardware, connect, etc…), since they will always be opened as normal webpages, they can have common chunks to reduce the overall size of the deployment bundle.

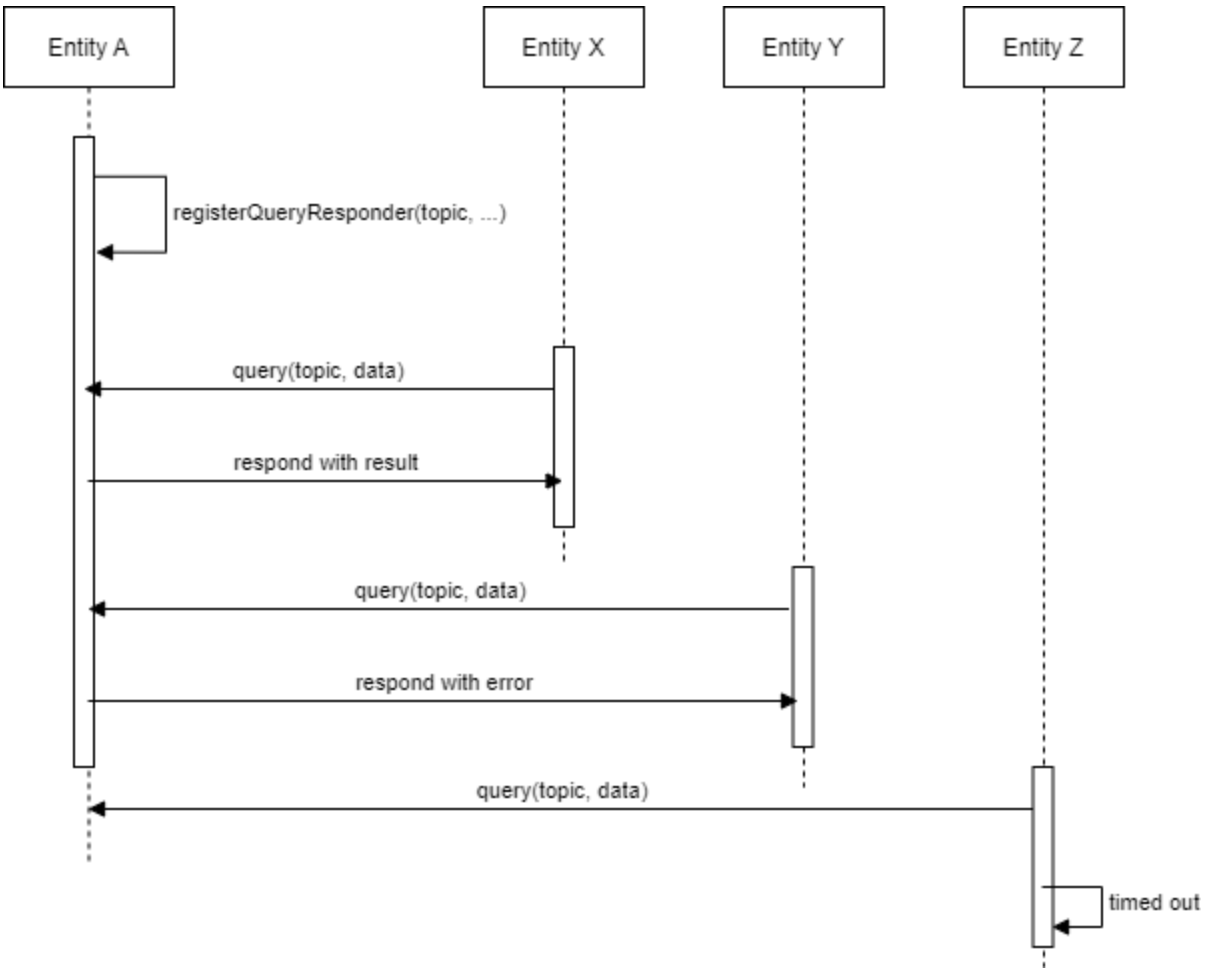
Communication Infrastructure

Since each entity has it's own isolated context without access to each others memory space, we are left with IPC messaging passing to propagate information from one entity to the other (with the exception of content_script inject_script)

The communication infrastructure is based on two communication patterns:

1. **Broadcast:** One entity sends a broadcast to every entity that might be listening on the topic of the broadcast. Other entities may choose to listen in on the communication channel for reception of such broadcasts specified by the topic.
2. **Querying:** One entity will register a responder for a topic. The responder can act upon a query received from another entity and optionally respond with a result. Other entities can therefore use that topic to query the responder.





Access Control

The patterns mentioned above also support access definition:

Broadcast: For broadcast, it's only possible to limit access when sending. We can send to all internals and/or to a list of tabs.

Query: Since the browser APIs allows us to detect the sender to some extend we can limit the access other entities can have to the responder. These accesses are defined by ACLs (Access Control List) we can limit the access to singleton entities (popup, service_worker), internal UI frames (connect, hardware), tabs interfaced through the content_script (all or specific tab Ids).

DOM Communication

Since communication between content_script and inject_script is mediated through the browser DOM, we have a different implementation for the patterns above when it comes to communication through the DOM. Almost everything is the same. However, they use slightly different naming and reside in a different folder entirely. Also since there are only two contexts involved (isolated content_script context and the website's untrusted context) having access control does not make sense.

Implementation

The methods for these patterns can be found here:

```
src/common/
  /messaging  ......  Has the required files and typings for broadcast
and query
  /dom  ............  Has the required files and types for dom based
communication
```

Communication Models

Atop of the patterns mentioned above, we have built higher level communication models that make the information propagation between entities more straightforward.

These models are divided mainly into three classes:

1. **State Provision:** This is used to define one entity as the provider of a certain state information, the provider is almost always the service_worker as it's the only entity that is long lived and there exists only one instance of. The other entities can then consume this state, they are able to query the initial state and be aware of all changes that occur via the broadcast pattern. Keep in mind that this model is unidirectional, only the provider can make changes to the state, the consumers are only ever able to read the state and be notified of change.

   The parts involved in making this model possible are as follows:

   Provider: The provider is a single instance of the `CommonStateProvider` class identified by the topic that is passed to it upon instantiation. Creating a provider requires the initial state so it can start provisioning as soon as it's setup, since the consumers will query for the current state as soon as they themselves are initialized.

   The Provider uses `immer`'s `produceWithPatches` whenever the owner entity tries to update the state. It then updates it's internal copy of the state and broadcasts the patches to the consumers so they can reconstruct the next state from their own copy of the current state and the patches returned by the produce function.

   Consumer: Consumers are instances of the `CommonStateConsumer` class. Any entity can have any arbitrary number consumers for one or any topic. Each consumer when initialized will request the current state and listen for updates. When updates happen it will update it's internal copy of the current state and inform the listeners via its events.

   A common use-case for these is to use them inside a react hook to provide the state to react components that rely on the represented state, a helper method is available for this exact purpose named `useStateConsumer` which can take in a `selector` to only update the using component only if the selected piece of the state is changed.

   ConsumerFragment: Fragments are helper instances of the `CommonStateConsumerFragment` class. It's easier to think of them as `useStateConsumer` for non-react usage. They acquire and listen on changes for a piece of the state articulated by the `selector` that is passed into them upon construction.

2. **Operation:** Working with operations is similar to RPC invocation. At the core, it's only a query, but in order to organize them and make them typesafe, we have decided to put them all under the same folder structure to ensure consistency and ease of maintenance. They almost always have a `registerResponder` method, and a `perform` method which correspond to `registerQueryResponder` and `query` but also holds the type definition and topic string in the same place.
3. **Event:** As Operations are wrappers around the query flow, events are wrappers around the broadcast flow. For the same reasons as mentioned above: typesafety, ease of maintenance and consistency.

DOM Communication Models

Since we have different communication patterns for DOM communication, we also have different models for DOM communication as well. These models tightly correspond to their internal model counterparts. Therefore, we will only mention their names and their slight differences:

1. **State Provision:** For DOM they are called `DOMStateProvider` and `DOMStateConsumer`. Fragment does not make sense in this context, also since there are only two contexts involved, initialization is much simpler.
2. **Operation:** Here they are referred to as `DOMActions`
3. **Event:** Referred to as `DOMEvents`

Website to Service Worker Communication Path

As you might have noticed, there is no direct communication path from the inject_script (untrusted website context) to the service_worker. In order to make it possible, we must use the content_script as a mediator.

For states, the content_script will consume the internal state and provide the relevant piece via the DOM State provision models, this is where `CommonStateConsumerFramgment` comes in handy, having this extra layer, ensures that we do not accidentally expose state that we do not wish to be exposed while also futureproofing it.

For operations and events, we must setup responders and listeners that **sanitize** and then forward the request/event to the service_wroker via the internal communication models.

Implementation

The communication models for both internal entities and DOM are found in the paths below:

```
/src/common
  /states  .........  Holds the code for all internal state provisions
    /channel  ......  The main Class defintions are held here
    /public  .......  Holds the public state defintions (state that
does not need security)
    /secure  .......  Holds secure states (require security audition
and consideration)
  /operations ......  All the internal operation definitions with their
ACLs are listed here
  /events  .........  Same as operations but for events
  /dom
    /states  .......  DOM state classes and definitions
    /actions  ......  Same as internal operations but for DOM
    /events  .......  Same as actions but for events
```

Entity Architectures

In this sections we will look at the internal architectures of some of the main entities. Since content_script and inject_script are heavily coupled we will treat them as one for our architectural analysis purposes.

It is important to mention that service_worker is at the hearth of the extension since it handles most of the state provisions, encryption services, securing secrets and much more. Therefore, we will start from this entity and the move onto popup and related frames such as connect and hardware (and the reason they exist in the first place). And we end this section with a brief overview of the content_script/inject_script duo.

Service Worker Architecture

WIP

Popup Frame Architecture

WIP

Connect and Hardware Overview

WIP

Content Script and Inject Script Overview

WIP