What’s Up With //content

With Special Guest John

Intro

What lives in the content directory? What is the content layer? How does it fit into Chrome and the web at large? On the program today is John, who not only is a Content owner, but actually split the codebase to create the Content layer.

What is the Content layer?

The term “content” is used a lot across Chrome and can refer to many different, but related things.
//content refers to the directory in the Chrome code base (here).
The **content layer** is the part of the Chrome codebase responsible for the multi-process sandbox implementation of the web platform.
The **Content public API** is a part of the content layer that embedders have access to, through which they use the content layer.
An **embedder** of the content layer is someone who uses the content layer to build a browser on top of that layer.

History

Chrome started small, but as its user base and code base grew, things got large and unwieldy. This led to the idea of separating the product from the platform. The product is all the stuff that makes Chrome unique, as a piece of software. The platform is what any minimal browser doing the latest HTML specs would need, to be sandboxed and multi process. Content is the platform portion of it. All the things you need for a browser that aren’t the UI.

At the start, the first embedder of Content was Chrome. As development continued, folks who worked on Content, rather than Chrome features, saw the need for a smaller binary that would be faster to build and debug. This led to content_shell, the second Content embedder.
What started as a code health improvement unexpectedly was very useful for other projects as a starting point for building a browser. Now that includes Android WebView, the Electron framework and the Chromium Embedded Framework, which build not only browsers but entire frameworks on top of Content. These other products have HTML data they need to render, and use Content, which is stable and secure, to help them do so.

Layering

Content is one of the many layers needed to build a browser. When you embed content, you're also including everything below the Content layer, such as Blink, V8, //net, etc. Chrome, as an embedder of Content sits on top of all of this.
That means Chrome can include files from the Content public API, but the reverse is not true. Content cannot include any files from //chrome, since other browsers and embedders can't depend on it. Anything in Content is used across all browsers. Chrome is only one such browser.
This leads to two directions of APIs coming from the Content layer. One being calls into Content, that embedders can call, and the other being APIs those embedders then have to implement.

Relationship with Blink

Blink is the rendering engine that Chrome used (formerly WebKit). That didn't have the concept of processes, but the browser is multiprocess, and data might be needed from outside the renderer sandbox. Content is the layer that wraps around the rendering engine, and uses the networking library and other things that provide a fully working browser.
Often, there are classes in the renderer processes that need to be driven from the browser process. That's where classes in the browser process with the suffix -Host come in. For example, every render process has a corresponding class in the browser process called RenderProcessHost. The renderer has tab objects called RenderView, with RenderViewHost in //content/browser.

Security

Content is the initial process that starts in the browser processes. It also creates renderer processes where Blink runs, as well as GPU processes and a networking process. Other processes include storage, audio and many other short lived processes for security and stability.

With all the processes needed for a browser, they communicate with each other via IPC (Mojo), which is a major security boundary. For example, there are multiple renderer processes per tab, which need to communicate with each other. When communicating between processes, they probably have different privilege levels, so any time you make a Mojo call, you have to be careful what information is being passed.
Embedders can also add their own IPCs, which introduce more possibilities for security vulnerabilities.
If a chain of vulnerabilities lead to the attacker having access to the browser process, that's very bad, as it's a highly privileged process. At that point, it's possible for an attacker to leak persistent changes to the user's system.

Using Content as an engineer

A typical Chrome engineer might interact with the Content API by needing something from the renderer when working in the browser, which would warrant adding a call or a parameter to an existing call in the Content public API.
If you're working within the browser process, you still may use the Content public API to get information about something happening elsewhere in the browser process. For example, information about a different tab.

WebContents class

At the start, every tab had a class to represent the content in that tab. That was called WebContents. It was called that because there were other contents like TabContents. Those have left, but WebContents remains.
Each WebContents roughly corresponds with one render process. There are more special types of WebContents now, like a WebContents within a WebContents.
WebContents keeps track of permissions for the tab, like audio/video, as well as navigation for the tab. It's also how the embedder can control a tab.

Content and the web

There are a lot of web standards that a browser needs to implement and do so correctly. This happens across the codebase, with some directories being particularly important.
One example is //net, which implements IETF and some web specs. V8 has to follow the EcmaScript standards, being a JavaScript engine.
For the web platform standards, some can be implemented in one process, so those may be done entirely in Blink. Some require multiple processes, or access to devices, so that implementation will be split across Blink and //content/browser.
To make sure this is done correctly and doesn't cause regressions, there are many tests to check that behaviour. Within Chrome there are Blink tests. Now there are also web platform tests that are shared across embedders that all the browsers can run to make sure the specs are met.

What (doesn't) belongs in the Content public API

More details on that here: https://crsrc.org/c/content/public/README.md

The Content public API is large and quite difficult to understand. As such, adding to the API is done carefully, only adding what's needed.
An example of what doesn't belong is using the Content layer as a tool for communicating between different parts of the Chrome layer. Similarly, we try to have only one way of doing a certain thing in the public API, to keep the already large API from getting unnecessarily larger.
Test only methods we also try to avoid, not only in the public API but production code in general.