Making a Downloader

introduction

Creating custom downloaders is only for advanced users who understand HTML or JSON. Beware! If you are simply looking for how to add new downloaders, please head over here.

this system

The first versions of hydrus's downloaders were all hardcoded and static--I wrote everything into the program itself and nothing was user-creatable or -fixable. After the maintenance burden of the entire messy system proved too large for me to keep up with and a semi-editable booru system proved successful, I decided to overhaul the entire thing to allow user creation and sharing of every component. It is designed to be very simple to the front-end user--they will typically handle a couple of png files and then select a new downloader from a list--but very flexible (and hence potentially complicated) on the back-end. These help pages describe the different compontents with the intention of making an HTML- or JSON- fluent user able to create and share a full new downloader on their own.

As always, this is all under active development. Your feedback on the system would be appreciated, and if something is confusing or you discover something in here that is out of date, please let me know.

what is a downloader?

In hydrus, a downloader is one of:

- ### Gallery Downloader
- This takes a string like 'blue_eyes' to produce a series of thumbnail gallery page URLs that can be parsed for image page URLs which can ultimately be parsed for file URLs and metadata like tags. Boorus fall into this category.
- ### URL Downloader
- This does just the Gallery Downloader's back-end--instead of taking a string query, it takes the gallery or post URLs directly from the user, whether that is one from a drag-and-drop event or hundreds pasted from clipboard. For our purposes here, the URL Downloader is a subset of the Gallery Downloader.

- ### Watcher
- This takes a URL that it will check in timed intervals, parsing it for new URLs that it then queues up to be downloaded. It typically stops checking after the 'file velocity' (such as '1 new file per day') drops below a certain level. It is mostly for watching imageboard threads.
- ### Simple Downloader
- This takes a URL one-time and parses it for direct file URLs. This is a miscellaneous system for certain simple gallery types and some testing/'I just need the third <img> tag's *src* on this one page' jobs.

The system currently supports HTML and JSON parsing. XML should be fine under the HTML parser--it isn't strict about checking types and all that.

what does a downloader do?

The Gallery Downloader is the most complicated downloader and uses all the possible components. In order for hydrus to convert our example 'blue_eyes' query into a bunch of files with tags, it needs to:

- Present some user interface named 'safebooru tag search' to the user that will convert their input of 'blue_eyes' into

  https://safebooru.org/index.php?page=post&s=list&tags=blue_eyes&pid=0.
- Recognise https://safebooru.org/index.php?page=post&s=list&tags=blue_eyes&pid=0 as a Safebooru Gallery URL.
- Convert the HTML of a Safebooru Gallery URL into a list URLs like

  https://safebooru.org/index.php?page=post&s=view&id=2437965 and possibly a 'next page' URL (e.g.

  https://safebooru.org/index.php?page=post&s=list&tags=blue_eyes&pid=40) that points to the next page of thumbnails.
- Recognise the https://safebooru.org/index.php?page=post&s=view&id=2437965 URLs as Safebooru Post URLs.
- Convert the HTML of a Safebooru Post URL into a file URL like

  https://safebooru.org//images/2329/b6e8c263d691d1c39a2eeba5e00709849d8f864d.jpg and some tags like: 1girl, bangs, black gloves, blonde hair, blue eyes, braid, closed mouth, day, fingerless gloves, fingernails, gloves, grass, hair ornament, hairclip, hands clasped, creator:hankuri, interlocked fingers, long hair, long sleeves, outdoors, own hands together, parted bangs, pointy ears, character:princess zelda, smile, solo, series:the legend of zelda, underbust.

So we have three components:

- **Gallery URL Generator (GUG):** faces the user and converts text input into initialising Gallery URLs.
- **URL Class:** identifies URLs and informs the client how to deal with them.
- **Parser:** converts data from URLs into hydrus-understandable metadata.

URL downloaders and watchers do not need the Gallery URL Generator, as their input *is* an URL. And simple downloaders also have an explicit 'just download it and parse it with this simple rule' action, so they do not use URL Classes (or even full-fledged Page Parsers) either.

gallery url generators

GUGs

Gallery URL Generators, or **GUGs** are simple objects that take a simple string from the user, like:

- blue_eyes
- blue_eyes blonde_hair
- InCase
- elsa dandon_fuga
- wlop
- goth* order:id_asc

And convert them into an initialising Gallery URL, such as:

- http://safebooru.org/index.php?page=post&s=list&tags=blue_eyes&pid=0
- https://konachan.com/post?page=1&tags=blonde_hair+blue_eyes
- https://www.hentai-foundry.com/pictures/user/InCase/page/1
- http://rule34.paheal.net/post/list/elsa dandon_fuga/1
- https://www.deviantart.com/wlop/favourites/?offset=0
- https://danbooru.donmai.us/posts?page=1&tags=goth*+order:id_asc

These are all the 'first page' of the results if you type or click-through to the same location on those sites. We are essentially emulating their own simple search-url generation inside the hydrus client.

actually doing it

Although it is usually a fairly simple process of just substituting the inputted tags into a string template, there are a couple of extra things to think about. Let's look at the ui under *network->downloader definitions->manage gugs*:

The client will split whatever the user enters by whitespace, so 'blue_eyes blonde_hair' becomes two *search terms*, ['blue_eyes', 'blonde_hair'], which are then joined back together with the given 'search terms separator', to make 'blue_eyes+blonde_hair'. Different sites use different separators, although ' ', '+', and ',' are most common. The new string is substituted into the '%tags%' in the template phrase, and the URL is made.

Note that you will not have to make %20 or %3A percent-encodings for reserved characters here--the network engine handles all that before the request is sent. For the most part, if you need to include or a user puts in ':' or ' ' or '□□□□', you can just pass it along straight into the final URL without worrying.

This ui should update as you change it, so have a play and look at how the output example url changes to get a feel for things. Look at the other defaults to see different examples. Even if you break something, you can just cancel out.

The name of the GUG is important, as this is what will be listed when the user chooses what 'downloader' they want to use. Make sure it has a clear unambiguous name.

The initial search text is also important. Most downloaders just take some text tags, but if your GUG expects a numerical artist id (like pixiv artist search does), you should specify that explicitly to the user. You can even put in a brief '(two tag maximum)' type of instruction if you like.

Notice that the Deviart Art example above is actually the stream of wlop's *favourites*, not his works, and without an explicit notice of that, a user could easily mistake what they have selected. 'gelbooru' or 'newgrounds' are bad names, 'type here' is a bad initialising text.

Nested GUGs

Nested Gallery URL Generators are GUGs that hold other GUGs. Some searches actually use more than one stream (such as a Hentai Foundry artist lookup, where you might want to get both their regular works and their scraps, which are two separate galleries under the site), so NGUGs allow you to generate multiple initialising URLs per input. You can experiment with this ui if you like--it isn't too complicated--but you might want to hold off doing anything for real until you are comfortable with everything and know how producing multiple initialising URLs is going to work in the actual downloader.

url classes

url classes

The fundamental connective tissue of the downloader system is the 'URL Class'. This object identifies and normalises URLs and links them to other components. Whenever the client handles a URL, it tries to match it to a URL Class to figure out what to do.

the types of url

For hydrus, an URL is useful if it is one of:

- ### File URL
- This returns the full, raw media file with no HTML wrapper. They typically end in a filename like

  http://safebooru.org//images/2333/cab1516a7eecf13c462615120ecf781116265f17.jpg, but sometimes they have a more complicated fetch command ending like 'file.php?id=123456' or '/post/content/123456'.
  These URLs are remembered for the file in the 'known urls' list, so if the client happens to encounter the same URL in future, it can determine whether it can skip the download because the file is already in the database or has previously been deleted.
  It is not important that File URLs be matched by a URL Class. File URL is considered the 'default', so if the client finds no match, it will assume the URL is a file and try to download and import the result. You might want to particularly specify them if you want to present them in the media viewer or discover File URLs are being confused for Post URLs or something.
- ### Post URL
- This typically return some HTML that contains a File URL and metadata such as tags and post time. They sometimes present multiple sizes (like 'sample' vs 'full size') of the file or even different formats (like 'ugoira' vs 'webm'). The Post URL for the file above,

  http://safebooru.org/index.php?page=post&s=view&id=2429668 has this 'sample' presentation. Finding the best File URL in these cases can be tricky!

This URL is also saved to 'known urls' and will usually be similarly skipped if it has previously been downloaded. It will also appear in the media viewer as a clickable link.

- ### Gallery URL
- This presents a list of Post URLs or File URLs. They often also present a 'next page' URL. It could be a page like http://safebooru.org/index.php?page=post&s=list&tags=yorha_no._2_type_b&pid=0 or an API URL like http://safebooru.org/index.php?page=dapi&s=post&tags=yorha_no._2_type_b&q=index&pid=0.

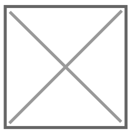
- ### Watchable URL
- This is the same as a Gallery URL but represents an ephemeral page that receives new files much faster than a gallery but will soon 'die' and be deleted. For our purposes, this typically means imageboard threads.

the components of a url

As far as we are concerned, a URL string has four parts:

- **Scheme:** "http" or "https"
- **Location/Domain:** "safebooru.org" or "i.4cdn.org" or "cdn002.somebooru.net"
- **Path Components:** "index.php" or "tesla/res/7518.json" or "pictures/user/daruak/page/2" or "art/Commission-animation-Elsa-and-Anna-541820782"
- **Query Parameters:** "page=post&s=list&tags=yorha_no._2_type_b&pid=40" or "page=post&s=view&id=2429668"

So, let's look at the 'edit url class' panel, which is found under *network->manage url classes*:



A TBIB File Page like https://tbib.org/index.php?page=post&s=view&id=6391256 is a Post URL. Let's look at the metadata first:

- ### Name and type
- Like with GUGs, we should set a good unambiguous name so the client can clearly summarise this url to the user. 'tbib file page' is good.

This is a Post URL, so we set the 'post url' type.

- ## Association logic

- All boorus and most sites only present one file per page, but some sites present multiple files on one page, usually several pages in a series/comic, as with pixiv. Danbooru-style thumbnail links to 'this file has a post parent' do not count here--I mean that a single URL embeds multiple full-size images, either with shared or separate tags. It is **very important** to the hydrus client's downloader logic (making decisions about whether it has previously visited a URL, so whether to skip checking it again) that if a site can present multiple files on a single page that 'can produce multiple files' is checked.
  Related is the idea of whether a 'known url' should be associated. Typically, this should be checked for Post and File URLs, which are fixed, and unchecked for Gallery and Watchable URLs, which are ephemeral and give different results from day to day. There are some unusual exceptions, so give it a brief thought--but if you have no special reason, leave this as the default for the url type.

And now, for matching the string itself, let's revisit our four components:

- ## Scheme

- TBIB supports http and https, so I have set the 'preferred' scheme to https. Any 'http' TBIB URL a user inputs will be automatically converted to https.

- ## Location/Domain

- For Post URLs, the domain is always "tbib.org".
  The 'allow' and 'keep' subdomains checkboxes let you determine if a URL with "artistname.artsite.com" will match a URL Class with "artsite.com" domain and if that subdomain should be remembered going forward. Most sites do not host content on subdomains, so you can usually leave 'match' unchecked. The 'keep' option (which is only available if 'keep' is checked) is more subtle, only useful for rare cases, and unless you have a special reason, you should leave it checked. (For keep: In cases where a site farms out File URLs to CDN servers on subdomains--like randomly serving a mirror of "https://muhbooru.org/file/123456" on "https://srv2.muhbooru.org/file/123456"--and removing the subdomain still gives a valid URL, you may not wish to keep the subdomain.) Since TBIB does not use subdomains, these options do not matter--we can leave both unchecked.
  'www' and 'www2' and similar subdomains are automatically matched. Don't worry about them.

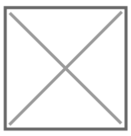
- ## Path Components

- TBIB just uses a single "index.php" on the root directory, so the path is not complicated. Were it longer (like "gallery/cgi/index.php", we would add more ("gallery" and "cgi"), and since the path of a URL has a strict order, we would need to arrange the items in the listbox there so they were sorted correctly.

- Query Parameters
- TBIB's index.php takes many query parameters to render different page types. Note that the Post URL uses "s=view", while TBIB Gallery URLs use "s=list". In any case, for a Post URL, "id", "page", and "s" are necessary and sufficient.

string matches

As you edit these components, you will be presented with the Edit String Match Panel:



This lets you set the type of string that will be valid for that component. If a given path or query component does not match the rules given here, the URL will not match the URL Class. Most of the time you will probably want to set 'fixed characters' of something like "post" or "index.php", but if the component you are editing is more complicated and could have a range of different valid values, you can specify just numbers or letters or even a regex pattern. If you try to do something complicated, experiment with the 'example string' entry to make sure you have it set how you think.

Don't go overboard with this stuff, though--most sites do not have super-fine distinctions between their different URL types, and hydrus users will not be dropping user account or logout pages or whatever on the client, so you can be fairly liberal with the rules.

how do they match, exactly?

This URL Class will be assigned to any URL that matches the location, path, and query. Missing path compontent or query parameters in the URL will invalidate the match but additonal ones will not!

For instance, given:

- URL A: https://8ch.net/tv/res/1002432.html
- URL B: https://8ch.net/tv/res
- URL C: https://8ch.net/tv/res/1002432
- URL D: https://8ch.net/tv/res/1002432.json
- URL Class that looks for "(characters)/res/(numbers).html" for the path

Only URL A will match

And:

- URL A: https://boards.4chan.org/m/thread/16086187
- URL B: https://boards.4chan.org/m/thread/16086187/ssg-super-sentai-general-651
- URL Class that looks for "(characters)/thread/(numbers)" for the path

Both URL A and B will match

And:

- URL A: https://www.pixiv.net/member_illust.php?mode=medium&illust_id=66476204
- URL B: https://www.pixiv.net/member_illust.php?mode=medium&illust_id=66476204&lang=jp
- URL C: https://www.pixiv.net/member_illust.php?mode=medium
- URL Class that looks for "illust_id=(numbers)" in the query

Both URL A and B will match, URL C will not

If multiple URL Classes match a URL, the client will try to assign the most 'complicated' one, with the most path components and then query parameters.

Given two example URLs and URL Classes:

- URL A: https://somebooru.com/post/123456
- URL B: https://somebooru.com/post/123456/manga_subpage/2
- URL Class A that looks for "post/(number)" for the path
- URL Class B that looks for "post/(number)/manga_subpage/(number)" for the path

URL A will match URL Class A but not URL Class B and so will receive A.

URL B will match both and receive URL Class B as it is more complicated.

This situation is not common, but when it does pop up, it can be a pain. It is usually a good idea to match exactly what you need--no more, no less.

normalising urls

Different URLs can give the same content. The http and https versions of a URL are typically the same, and:

- https://gelbooru.com/index.php?page=post&s=view&id=3767497
- gives the same as:

- https://gelbooru.com/index.php?id=3767497&page=post&s=view

And:

- https://e621.net/post/show/1421754/abstract_background-animal_humanoid-blush-brown_ey
- is the same as:
- https://e621.net/post/show/1421754
- *is the same as:*
- https://e621.net/post/show/1421754/help_computer-made_up_tags-REEEEEEEE

Since we are in the business of storing and comparing URLs, we want to 'normalise' them to a single comparable beautiful value. You see a preview of this normalisation on the edit panel. Normalisation happens to all URLs that enter the program.

Note that in e621's case (and for many other sites!), that text after the id is purely decoration. It can change when the file's tags change, so if we want to compare today's URLs with those we saw a month ago, we'd rather just be without it.

On normalisation, all URLs will get the preferred http/https switch, and their query parameters will be alphabetised. File and Post URLs will also cull out any surplus path or query components. This wouldn't affect our TBIB example above, but it will clip the e621 example down to that 'bare' id URL, and it will take any surplus 'lang=en' or 'browser=netscape_24.11' garbage off the query text as well. URLs that are not associated and saved and compared (i.e. normal Gallery and Watchable URLs) are not culled of unmatched path components or query parameters, which can sometimes be useful if you want to match (and keep intact) gallery URLs that might or might not include an important 'sort=desc' type of parameter.

Since File and Post URLs will do this culling, be careful that you not leave out anything important in your rules. Make sure what you have is both necessary (nothing can be removed and still keep it valid) and sufficient (no more needs to be added to make it valid). It is a good idea to try pasting the 'normalised' version of the example URL into your browser, just to check it still works.

'default' values

Some sites present the first page of a search like this:

https://danbooru.donmai.us/posts?tags=skirt

But the second page is:

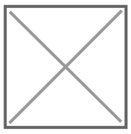
https://danbooru.donmai.us/posts?tags=skirt&page=2

Another example is:

https://www.hentai-foundry.com/pictures/user/Mister69M

https://www.hentai-foundry.com/pictures/user/Mister69M/page/2

What happened to 'page=1' and '/page/1'? Adding those '1' values in works fine! Many sites, when an index is absent, will secretly imply an appropriate 0 or 1. This looks pretty to users looking at a browser address bar, but it can be a pain for us, who want to match both styles to one URL Class. It would be nice if we could recognise the 'bare' initial URL and fill in the '1' values to coerce it to the explicit, automation-friendly format. Defaults to the rescue:



After you set a path component or query parameter String Match, you will be asked for an optional 'default' value. You won't want to set one most of the time, but for Gallery URLs, it can be hugely useful--see how the normalisation process automatically fills in the missing path component with the default! There are plenty of examples in the default Gallery URLs of this, so check them out. Most sites use page indices starting at '1', but Gelbooru-style imageboards use 'pid=0' file index (and often move forward 42, so the next pages will be 'pid=42', 'pid=84', and so on, although others use deltas of 20 or 40).

can we predict the next gallery page?

Now we can harmonise gallery urls to a single format, we can predict the next gallery page! If, say, the third path component or 'page' query parameter is always a number referring to page, you can select this under the 'next gallery page' section and set the delta to change it by. The 'next gallery page url' section will be automatically filled in. This value will be consulted if the parser cannot find a 'next gallery page url' from the page content.

It is neat to set this up, but I only recommend it if you actually cannot reliably parse a next gallery page url from the HTML later in the process. It is neater to have searches stop naturally because the parser said 'no more gallery pages' than to have hydrus always one page beyond and end every single search on an uglier 'No results found' or 404 result.

Unfortunately, some sites will either not produce an easily parsable next page link or randomly just not include it due to some issue on their end (Gelbooru is a funny example of this). Also, APIs will often have a kind of 'start=200&num=50', 'start=250&num=50' progression but not include that state in the XML or JSON they return. These cases require the automatic next gallery page rules (check out Artstation and tumblr api gallery page URL Classes in the defaults for examples of this).

how do we link to APIs?

If you know that a URL has an API backend, you can tell the client to use that API URL when it fetches data. The API URL needs its own URL Class.

To define the relationship, click the "String Converter" button, which gives you this:

You may have seen this panel elsewhere. It lets you convert a string to another over a number of transformation steps. The steps can be as simple as adding or removing some characters or applying a full regex substitution. For API URLs, you are mostly looking to isolate some unique identifying data ("m/thread/16086187" in this case) and then substituting that into the new API path. It is worth testing this with several different examples!

When the client links regular URLs to API URLs like this, it will still associate the human-pretty regular URL when it needs to display to the user and record 'known urls' and so on. The API is just a quick lookup when it actually fetches and parses the respective data.

parsers

parsers

In hydrus, a parser is an object that takes a single block of HTML or JSON data and returns many kinds of hydrus-level metadata.

Parsers are flexible and potentially quite complicated. You might like to open *network->manage parsers* and explore the UI as you read these pages. Check out how the default parsers already in the client work, and if you want to write a new one, see if there is something already in there that is similar--it is usually easier to duplicate an existing parser and then alter it than to create a new one from scratch every time.

There are three main components in the parsing system (click to open each component's help page):

- **Formulae:** Take parsable data, search it in some manner, and return 0 to n strings.
- **Content Parsers:** Take parsable data, apply a formula to it to get some strings, and apply a single metadata 'type' and perhaps some additional modifiers.
- **Page Parsers:** Take parsable data, apply content parsers to it, and return all the metadata in an appropriate structure.

Once you are comfortable with these objects, you might like to check out these walkthroughs, which create full parsers from nothing:

- e621 HTML gallery page
- Gelbooru HTML file page
- Artstation JSON file page API

Once you are comfortable with parsers, and if you are feeling brave, check out how the default imageboard and pixiv parsers work. These are complicated and use more experimental areas of the code to get their job done. If you are trying to get a new imageboard parser going and can't figure out subsidiary page parsers, send me a mail or something and I'll try to help you out!

When you are making a parser, consider this checklist (you might want to copy/have your own version of this somewhere):

- Do you get good URLs with good priority? Do you ever accidentally get favourite/popular/advert results you didn't mean to?

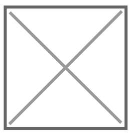
- If you need a next gallery page URL, is it ever not available (and hence needs a URL Class fix)? Does it change for search tags with unicode or http-restricted characters?
- Do you get nice namespaced tags? Are any unwanted single characters like -/+/? getting through?
- Is the file hash available anywhere?
- Is a source/post time available?
- Is a source URL available? Is it good quality, or does it often just point to an artist's base twitter profile? If you pull it from text or a tooltip, is it clipped for longer URLs?

putting it all together

putting it all together

Now you know what GUGs, URL Classes, and Parsers are, you should have some ideas of how URL Classes could steer what happens when the downloader is faced with an URL to process. Should a URL be imported as a media file, or should it be parsed? If so, how?

You may have noticed in the Edit GUG ui that it lists if a current URL Class matches the example URL output. If the GUG has no matching URL Class, it won't be listed in the main 'gallery selector' button's list--it'll be relegated to the 'non-functioning' page. Without a URL Class, the client doesn't know what to do with the output of that GUG. But if a URL Class does match, we can then hand the result over to a parser set at *network->downloader definitions->manage url class links*:



Here you simply set which parsers go with which URL Classes. If you have URL Classes that do not have a parser linked (which is the default for new URL Classes), you can use the 'try to fill in gaps...' button to automatically fill the gaps based on guesses using the parsers' example URLs. This is usually the best way to line things up unless you have multiple potential parsers for that URL Class, in which case it'll usually go by the parser name earliest in the alphabet.

If the URL Class has no parser set or the parser is broken or otherwise invalid, the respective URL's file import object in the downloader or subscription is going to throw some kind of error when it runs. If you make and share some parsers, the first indication that something is wrong is going to be several users saying 'I got this error: (*copy notes* from file import status window)'. You can then load the parser back up in *manage parsers* and try to figure out what changed and roll out an update.

manage url class links also shows 'api link review', which summarises which URL Classes api-link to others. In these cases, only the api URL gets a parser entry in the first 'parser links' window, since the first will never be fetched for parsing (in the downloader, it will always be converted to the API URL, and *that* is fetched and parsed).

Once your GUG has a URL Class and your URL Classes have parsers linked, test your downloader! Note that Hydrus's URL drag-and-drop import uses URL Classes, so if you don't have the GUG and gallery stuff done but you have a Post URL set up, you can test that just by dragging a Post URL from your browser to the client, and it should be added to a new URL Downloader and just work. It feels pretty good once it does!

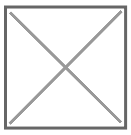
sharing downloaders

sharing

If you are working with users who also understand the downloader system, you can swap your GUGs, URL Classes, and Parsers separately using the import/export buttons on the relevant dialogs, which work in pngs and clipboard text.

But if you want to share conveniently, and with users who are not familiar with the different downloader objects, you can package everything into a single easy-import png as per here.

The dialog to use is *network->downloader definitions->export downloaders*:



It isn't difficult. Essentially, you want to bundle enough objects to make one or more 'working' GUGs at the end. I recommend you start by just hitting 'add gug', which--using Example URLs--will attempt to figure out everything you need by itself.

This all works on Example URLs and some domain guesswork, so make sure your url classes are good and the parsers have correct Example URLs as well. If they don't, they won't all link up neatly for the end user. If part of your downloader is on a different domain to the GUGs and Gallery URLs, then you'll have to add them manually. Just start with 'add gug' and see if it looks like enough.

Once you have the necessary and sufficient objects added, you can export to png. You'll get a similar 'does this look right?' summary as what the end-user will see, just to check you have everything in order and the domains all correct. If that is good, then make sure to give the png a sensible filename and embellish the title and description if you need to. You can then send/post that png wherever, and any regular user will be able to use your work.

login manager

login

The system works, but this help was never done! Check the defaults for examples of how it works, sorry!