Supplementary Document

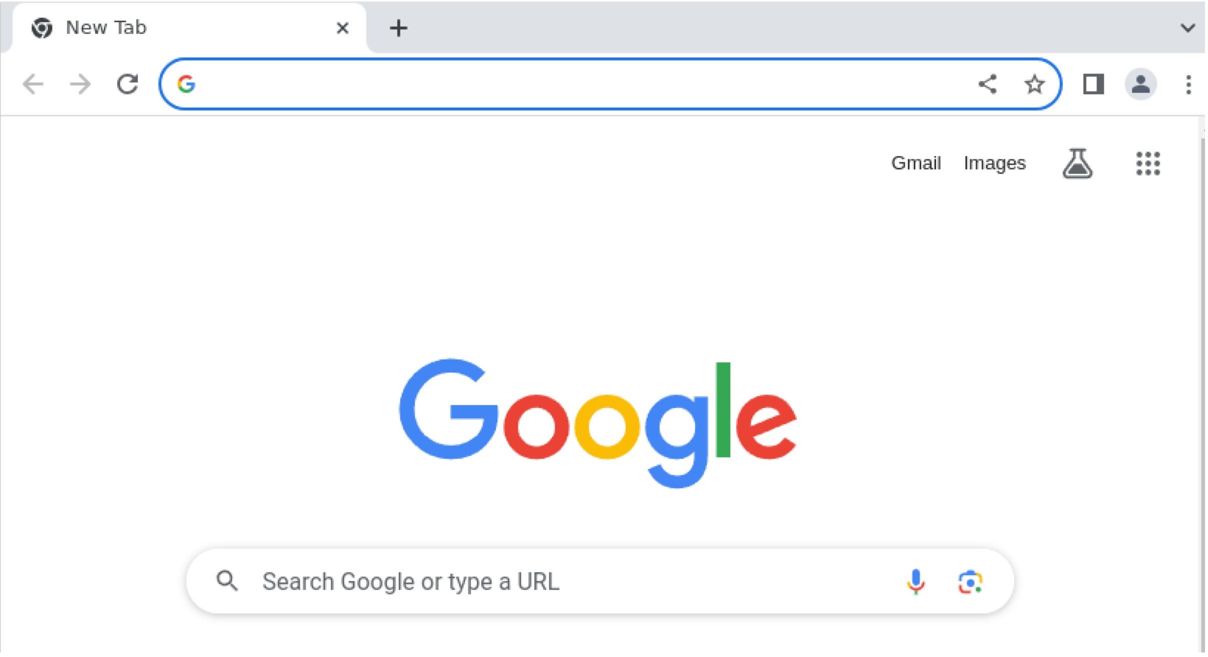
Outline

Causality Graph Model

Useful Resource: https://chromedevtools.github.io/devtools-protocol/

Frame

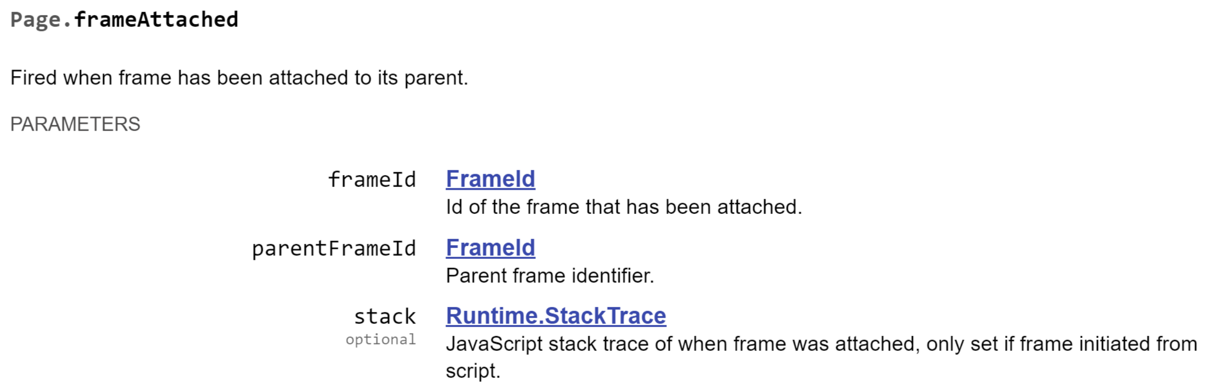
Let's walk through how the browser works. When you open a Chrome browser, you will first see a window (or tab) named "New Tab" displaying a Google home page.



Under the hood, the Google home page is displayed in a **frame**. You can consider a frame as a “content displayer”. A webpage typically consists of one top frame, where the main content of the page is displayed. In our example above, this base frame is sufficient to display the content of the Google home page.

Frame Attach and Navigation

Now how about those fancy webpages we see that display content from various sources (e.g., a webpage that has videos from YouTube and advertisements from Google)? In this case, to display content from multiple sources, the base frame will have to **attach** and **navigate** (these two operations are usually coupled together) to multiple frames which in turn are responsible for displaying the corresponding content. Here we see obvious relationships: **Frame — attach→ Frame** and **Frame — navigate→ Frame**. When attachment happens, an event **Page.frameAttached** carrying information of **frameId** and **parentFrameId**, and an optional field **stack** is emitted. If the optional **stack** field is present, it means that the frame is created by a script (see **Script** section below).



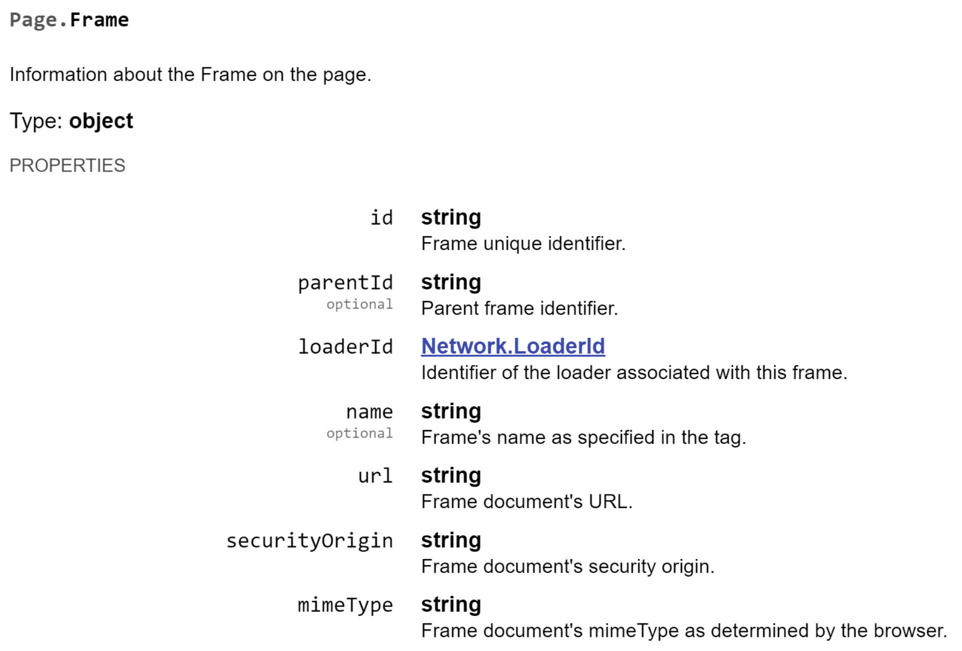
`Page.frameAttached`

Fired when frame has been attached to its parent.

PARAMETERS

`frameId`	FrameId Id of the frame that has been attached.
`parentFrameId`	FrameId Parent frame identifier.
`stack` optional	Runtime.StackTrace JavaScript stack trace of when frame was attached, only set if frame initiated from script.

When navigation happens, an event **Page.frameNavigated** carrying information including the child frame ID (**id**) and the parent frame ID (**parentId**), etc. will be emitted and captured by your auditor.



`Page.Frame`

Information about the Frame on the page.

Type: **object**

PROPERTIES

`id`	**string** Frame unique identifier.
`parentId` optional	**string** Parent frame identifier.
`loaderId`	**Network.LoaderId** Identifier of the loader associated with this frame.
`name` optional	**string** Frame's name as specified in the tag.
`url`	**string** Frame document's URL.
`securityOrigin`	**string** Frame document's security origin.
`mimeType`	**string** Frame document's mimeType as determined by the browser.

Then how about the first frame in a newly opened tab? Does it have a parent frame? The answer is no as you probably have figured out, and that’s why the **parentId** is an optional field. In this case, the **Page.frameNavigated** event emitted during the first frame navigation in a tab does not have the **parentId** field. Therefore, we can initialize **current_frame** as such a frame.

Loader

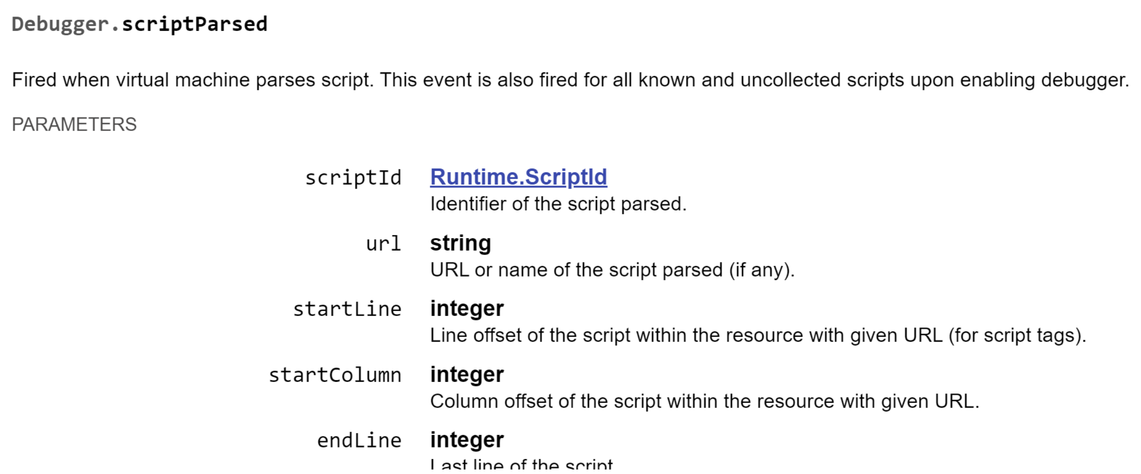
Now let's step back, what is responsible for loading the content then? The answer is the **loader**. A loader is a component in your browser responsible for loading and displaying web resources of various types, such as HTML (which contains frames), CSS, JavaScript, images, and others. Here we see an obvious relationship: Loader —load→ Frame**.**

Note that 1) a loader can load more than one frame (when a webpage has multiple frames to display different content), and 2) the same frame can be loaded by different loaders (when you have multiple tab open displaying the same webpage). Therefore, both loader ID and frame ID are required to uniquely identify a frame. Hence, we use **{frame_id}_{loader_id}** as the frame ID in our model.

Note that the new frame is navigated from the current frame (the parent frame). To maintain this relationship, intuitively enough, we should maintain a variable to store the current frame and establish a navigation edge from the current frame to the new frame. Now you may wonder, how about the blank page? We can identify this case by the fact that the **parentId** field would not be provided in the event message. When this happens, we set such a frame as the current frame.

Script

A nice feature of webpages is their ability to dynamically execute scripts to provide versatile functionalities and a better user experience. Attackers love to alter the scripts to be executed by a webpage to malicious ones to compromise users, for example, stealing their keystrokes typed into the webpage. Therefore, we need to model the events related to scripts. Each script has a **scriptId** which can uniquely identify a script in its owner frame. When a script gets executed by the current frame, a **Debugger.scriptParsed** event occurs, based on which we can create a script node and establish a parse edge between the current frame and the script Frame: **Frame — compile→ Script.**



`Debugger.scriptParsed`

Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

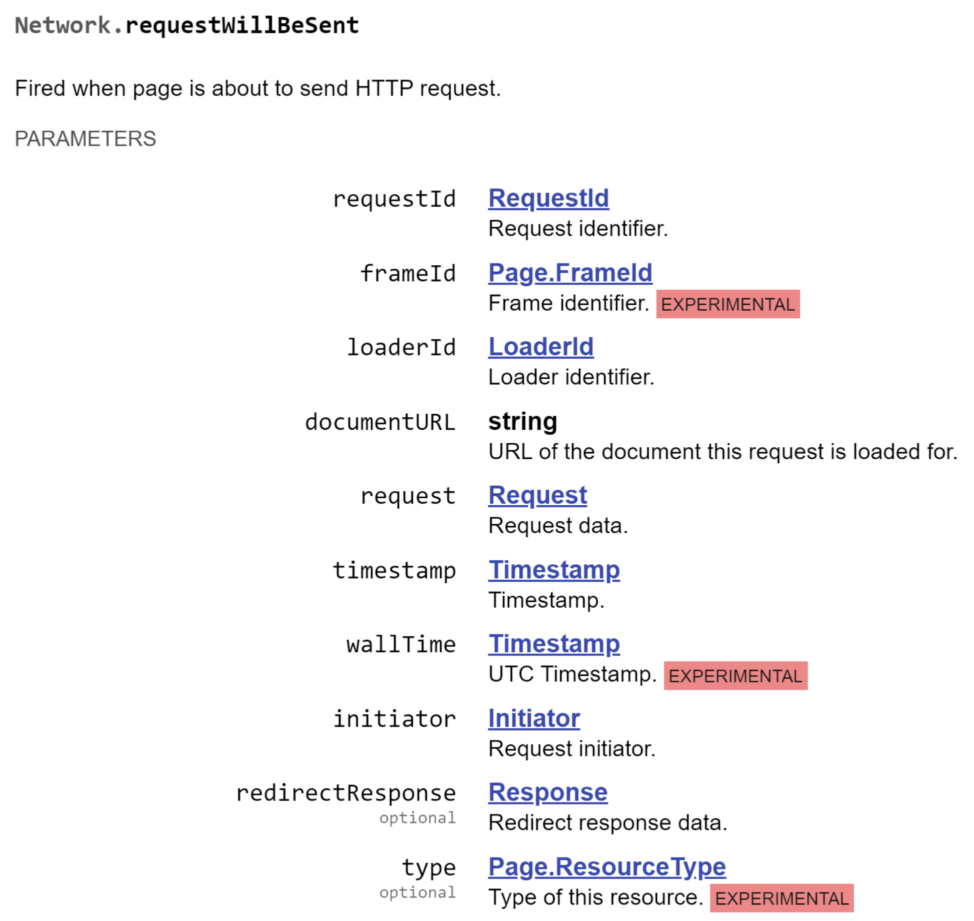
PARAMETERS

`scriptId`	Runtime.ScriptId Identifier of the script parsed.
`url`	**string** URL or name of the script parsed (if any).
`startLine`	**integer** Line offset of the script within the resource with given URL (for script tags).
`startColumn`	**integer** Column offset of the script within the resource with given URL.
`endLine`	**integer** Last line of the script

A script can be used to create frames (iFrames), request resources, open windows, etc. For example, a script can create an iFrame displaying a phishing webpage that covers the existing frame. Here we see another two relationships **Script — create→ Frame** and **Script — request→ Resource** where the resource is needed to display the new webpage in the new Frame.

Resource Requests & Responses

When you open a webpage, requests to various resources (images, videos, scripts, etc.) will be sent out so that you get the content displayed to you.



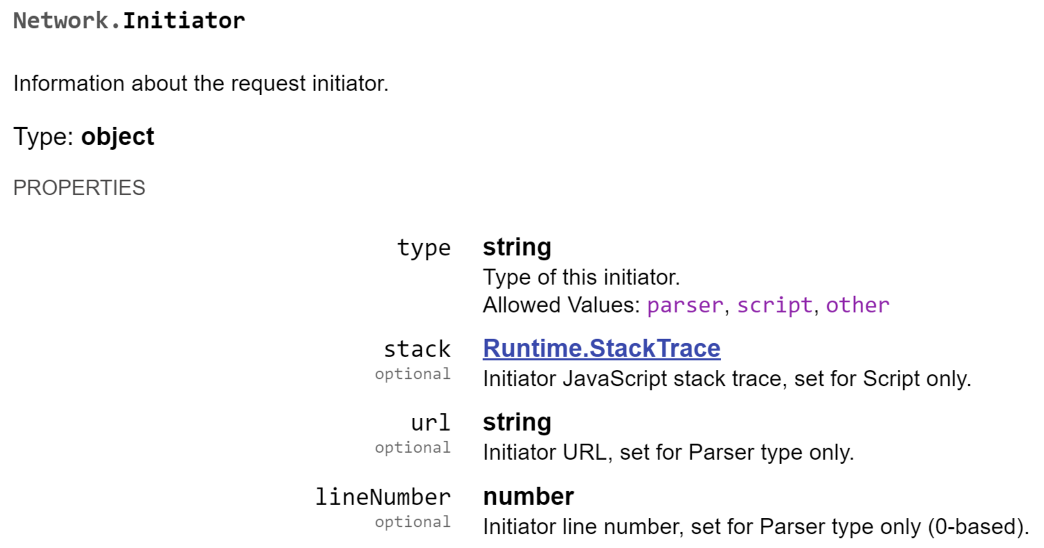
`Network.requestWillBeSent`

Fired when page is about to send HTTP request.

PARAMETERS

requestId	**RequestId** Request identifier.
frameId	**Page.FrameId** Frame identifier. EXPERIMENTAL
loaderId	**LoaderId** Loader identifier.
documentURL	**string** URL of the document this request is loaded for.
request	**Request** Request data.
timestamp	**Timestamp** Timestamp.
wallTime	**Timestamp** UTC Timestamp. EXPERIMENTAL
initiator	**Initiator** Request initiator.
redirectResponse optional	**Response** Redirect response data.
type optional	**Page.ResourceType** Type of this resource. EXPERIMENTAL

Note that those requests can be initiated by the parser of the current frame as well as scripts. We can find this information in the initiator type. **Parser** here is the HTML parser of a frame, indicating the request is sent from a frame. **Script** means that the request is sent from a script.



`Network.Initiator`

Information about the request initiator.

Type: **object**

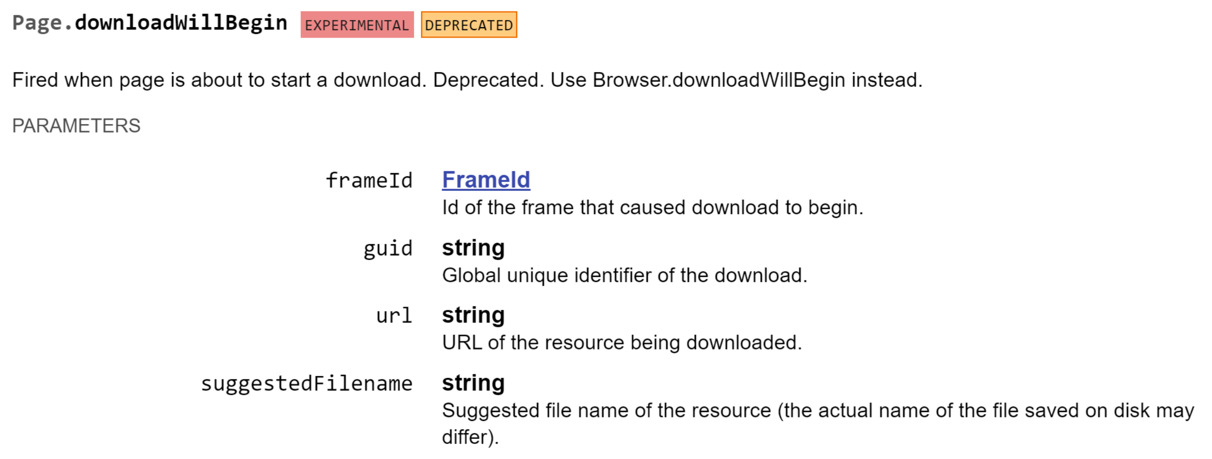
PROPERTIES

type	**string** Type of this initiator. Allowed Values: `parser`, `script`, `other`
stack optional	**Runtime.StackTrace** Initiator JavaScript stack trace, set for Script only.
url optional	**string** Initiator URL, set for Parser type only.
lineNumber optional	**number** Initiator line number, set for Parser type only (0-based).

Note that if the initiator is a script, the script should already be captured and added to our graph when **Debugger.scriptParsed** event occurs. Therefore, we just need to retrieve the corresponding script node from a data structure we maintain to keep track of parsed scripts as the initiator node. Now that we have figured out the initiator node, we just need to create a resource node using its URL as its identifier and add a request edge between the initiator (Frame or Script) and the resource. The relationships we can model here are **Frame/Script— request→ Resource** and **Resource — respond→ Frame/Script.** (Handling responses is just the reverse of handling requests.)

Download

A common web attack is to lure users into downloading malware to compromise their machines. When a download task starts (in a frame), the event **Page.downloadWillBegin** is emitted. To model the event in our graph, we can find the frame node based on its frame ID from a data structure map we maintain where each frame ID is mapped to the latest frame node created with this frame ID. The relationship we see here is rather simple: **Frame — download→ File**



`Page.downloadWillBegin` EXPERIMENTAL DEPRECATED

Fired when page is about to start a download. Deprecated. Use Browser.downloadWillBegin instead.

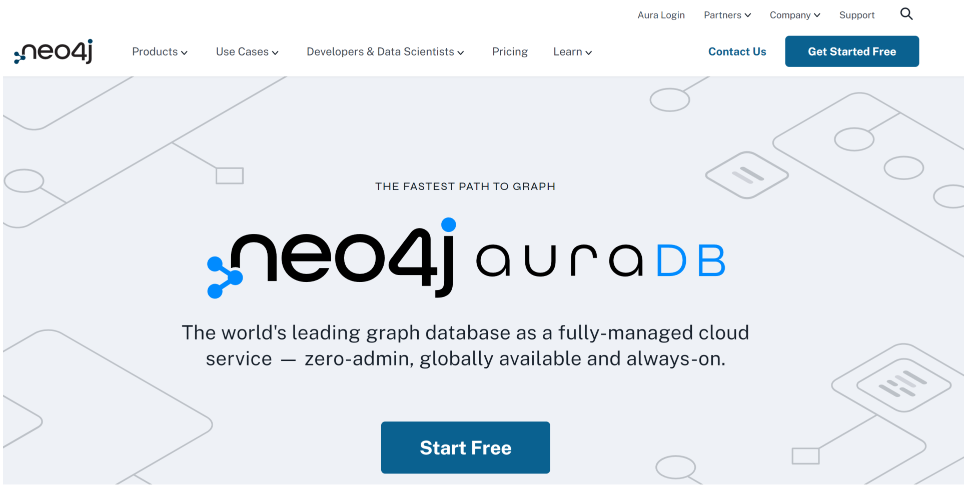
PARAMETERS

`frameId`	FrameId Id of the frame that caused download to begin.
`guid`	**string** Global unique identifier of the download.
`url`	**string** URL of the resource being downloaded.
`suggestedFilename`	**string** Suggested file name of the resource (the actual name of the file saved on disk may differ).

neo4j

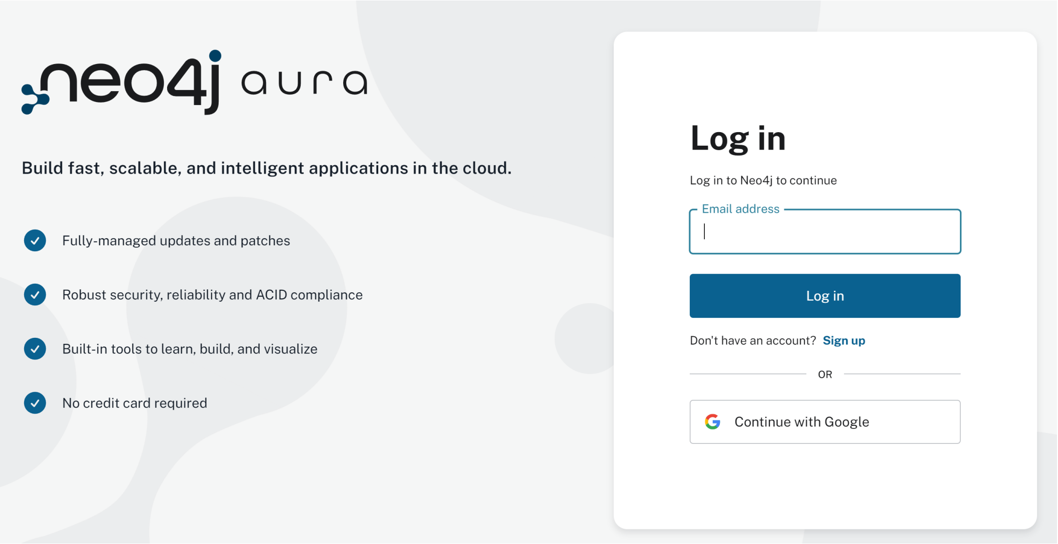
Construct Graphs

1. Open The neo4j cloud DB service https://neo4j.com/cloud/platform/aura-graph-database/?ref=nav-get-started-cta



The world's leading graph database as a fully-managed cloud service — zero-admin, globally available and always-on.

2. Click on `Start Free` and log in (I just choose to Continue with Google to avoid signing up)



Build fast, scalable, and intelligent applications in the cloud.

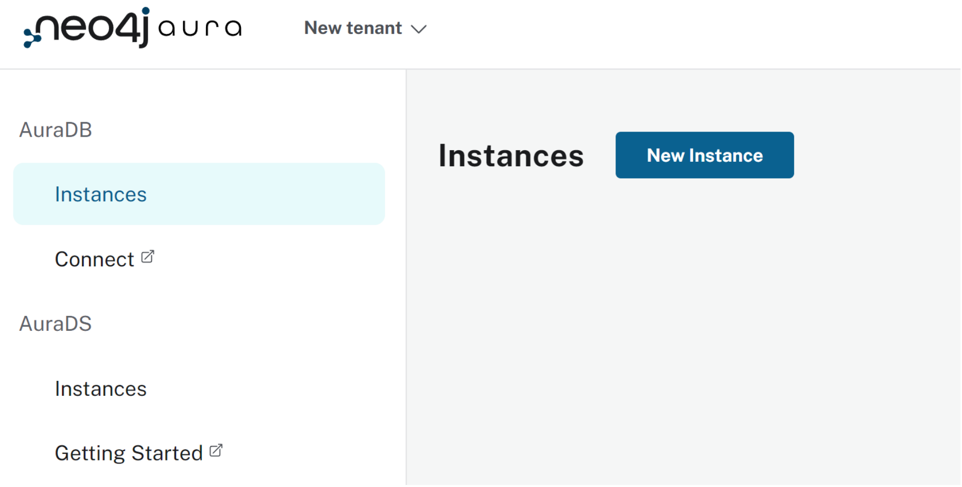
- ✓ Fully-managed updates and patches
- ✓ Robust security, reliability and ACID compliance
- ✓ Built-in tools to learn, build, and visualize
- ✓ No credit card required

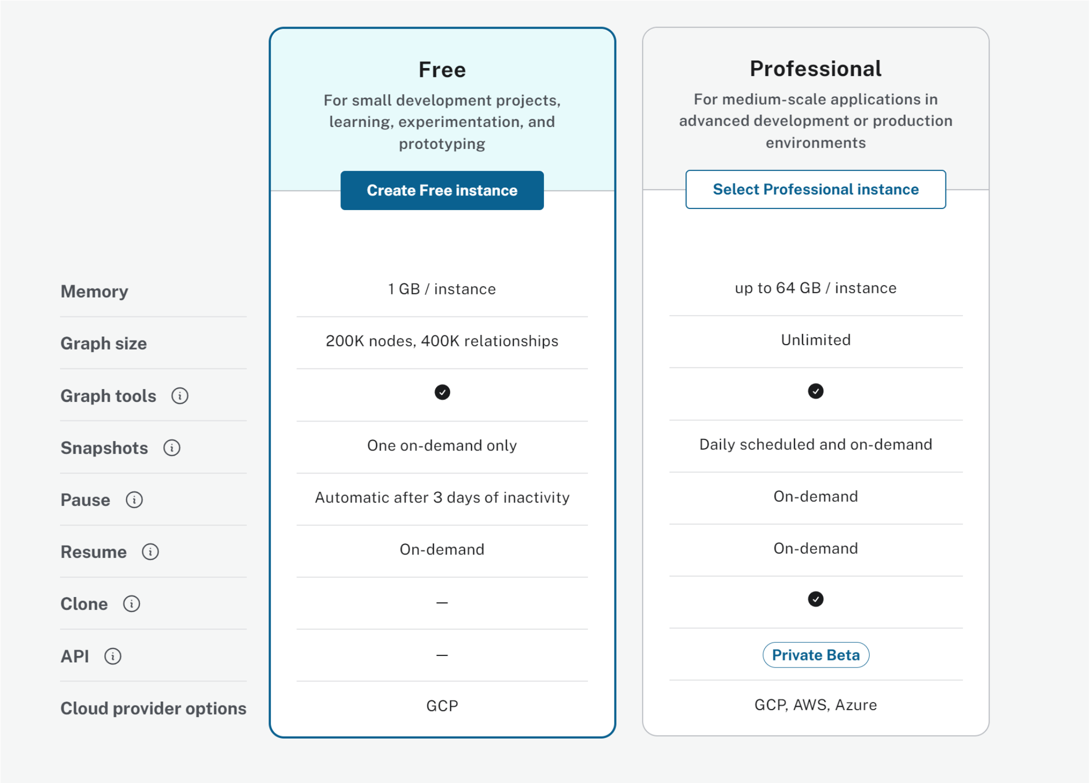
Log in

Log in to Neo4j to continue

Don't have an account? Sign up

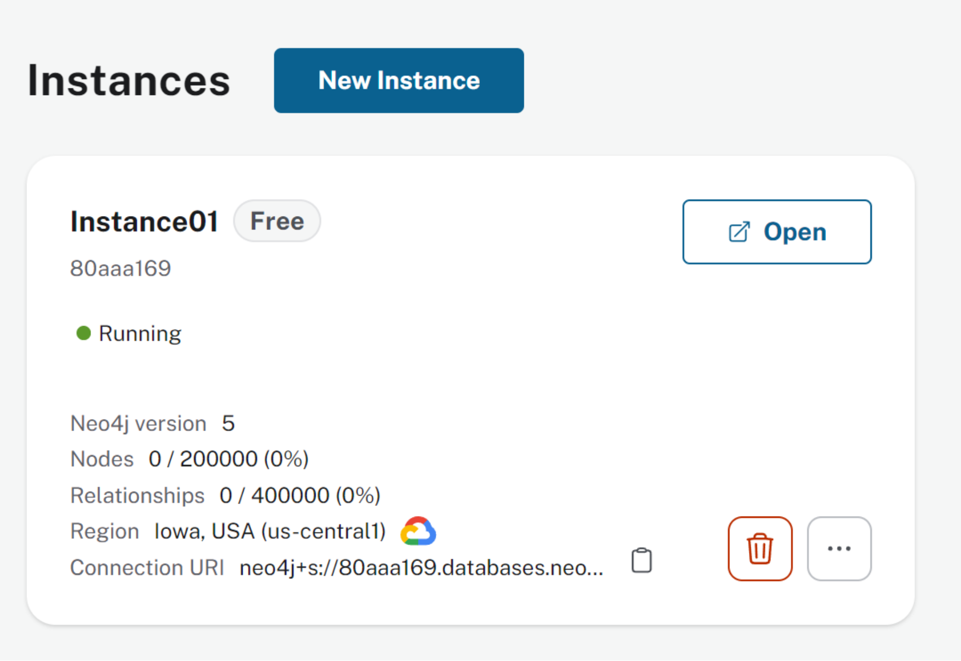
3. Once entering the panel, click on `New Instance` and choose the `Create Free instance`.





	Free	Professional
	For small development projects, learning, experimentation, and prototyping	For medium-scale applications in advanced development or production environments
	Create Free instance	Select Professional instance
Memory	1 GB / instance	up to 64 GB / instance
Graph size	200K nodes, 400K relationships	Unlimited
Graph tools ⓘ	✓	✓
Snapshots ⓘ	One on-demand only	Daily scheduled and on-demand
Pause ⓘ	Automatic after 3 days of inactivity	On-demand
Resume ⓘ	On-demand	On-demand
Clone ⓘ	—	✓
API ⓘ	—	Private Beta
Cloud provider options	GCP	GCP, AWS, Azure

4. An instance password will then be given to you. You can copy the password and download a copy. Then the instance will be created for you. Click on `Open`.



Instances

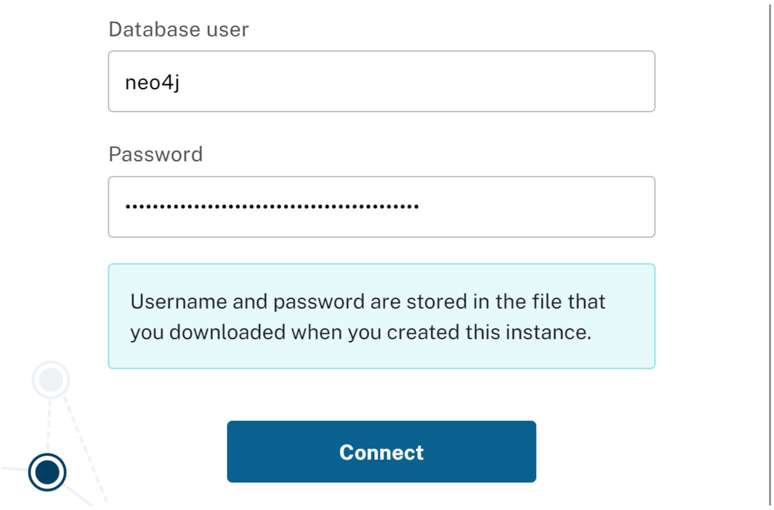
Instance01 Free

80aaa169

● Running

Neo4j version 5
Nodes 0 / 200000 (0%)
Relationships 0 / 400000 (0%)
Region Iowa, USA (us-central1)
Connection URI neo4j+s://80aaa169.databases.neo...

5. Paste the password and hit `Connect`.

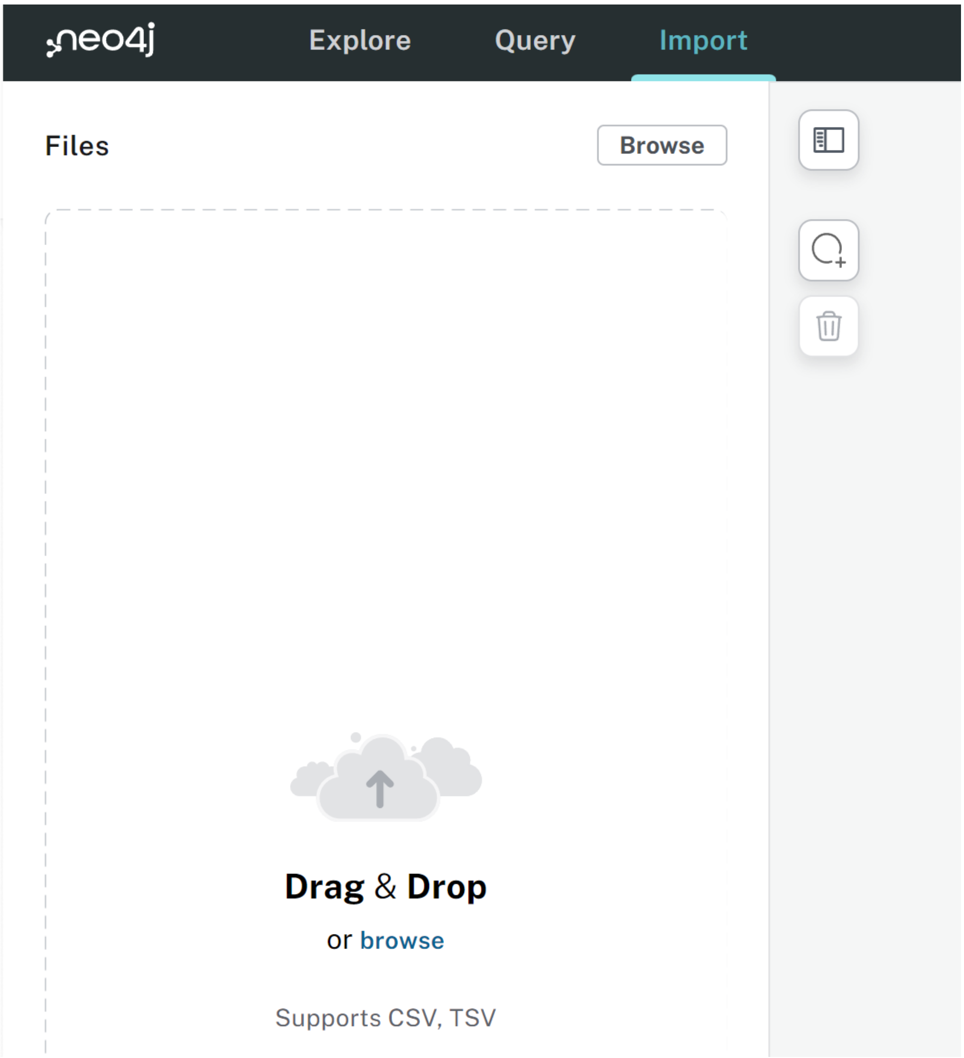


Database user: neo4j

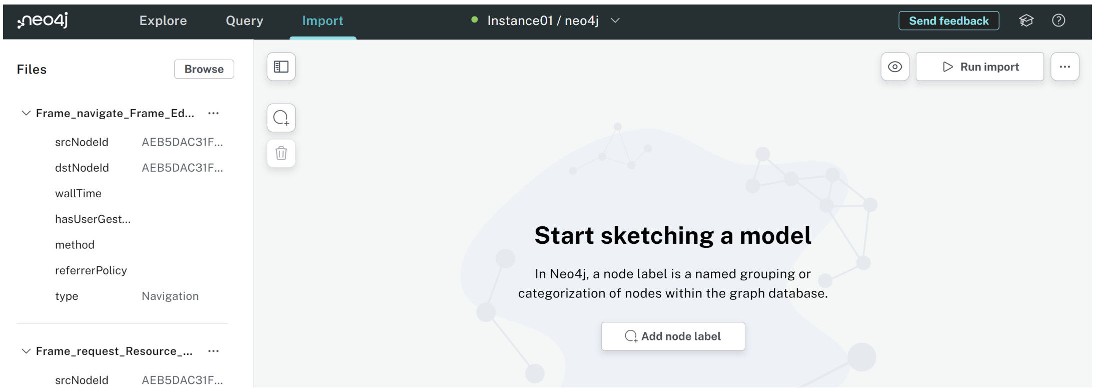
Password: ••

Username and password are stored in the file that you downloaded when you created this instance.

6. Navigate to `Import` in the panel and upload the `.tsv` log files under the `logs/` directory of the auditor.



7. Then click on `Add node label` and a circle will show up.



Frame_navigate_Frame_Ed...

srcNodeId AEB5DAC31F...
dstNodeId AEB5DAC31F...
wallTime
hasUserGest...
method
referrerPolicy
type Navigation

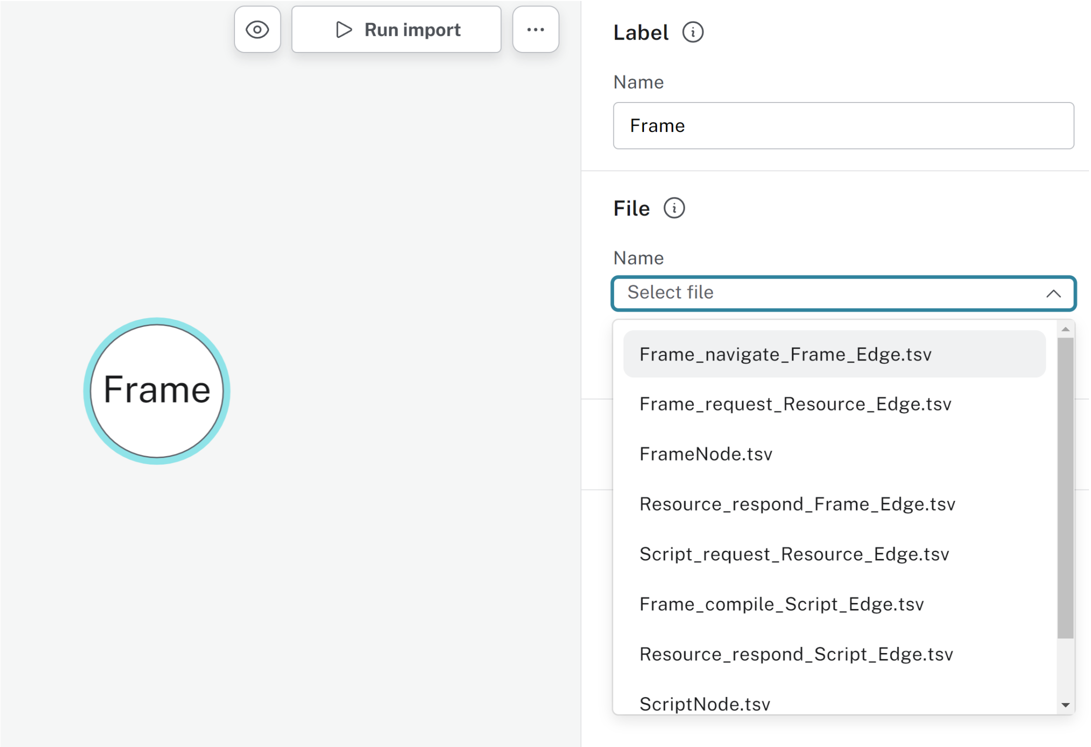
Frame_request_Resource_...

srcNodeId AEB5DAC31F...

Start sketching a model

In Neo4j, a node label is a named grouping or categorization of nodes within the graph database.

8. Type in the node name (e.g., `Frame`), and select the corresponding node file (e.g., `FrameNode.tsv`). Note: please set the first letter to be upper case (i.e., `Frame` instead of `frame`)



Label

Name

Frame

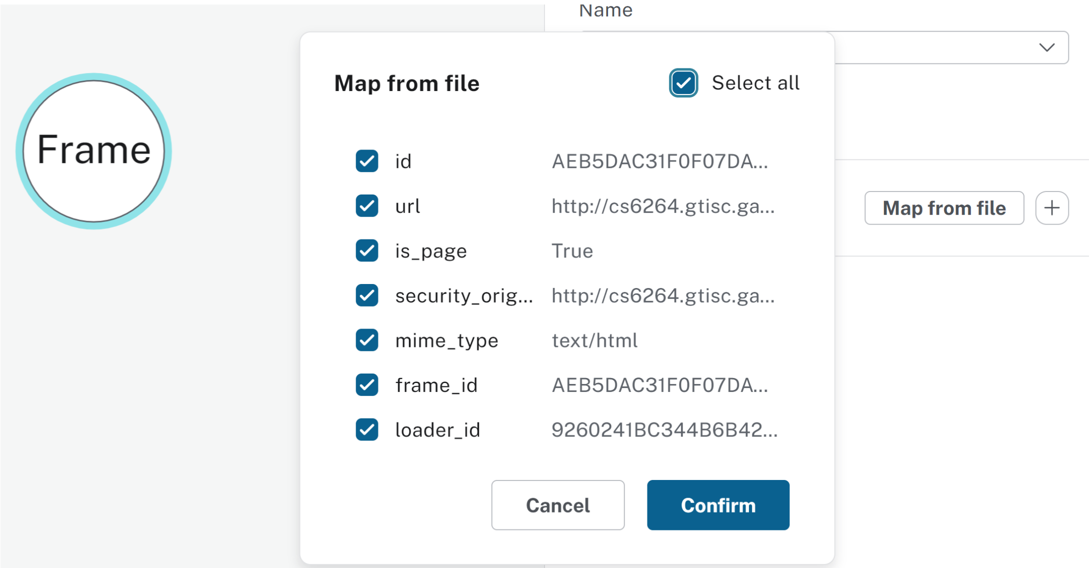
File

Name

Select file

- Frame_navigate_Frame_Edge.tsv
- Frame_request_Resource_Edge.tsv
- FrameNode.tsv
- Resource_respond_Frame_Edge.tsv
- Script_request_Resource_Edge.tsv
- Frame_compile_Script_Edge.tsv
- Resource_respond_Script_Edge.tsv
- ScriptNode.tsv

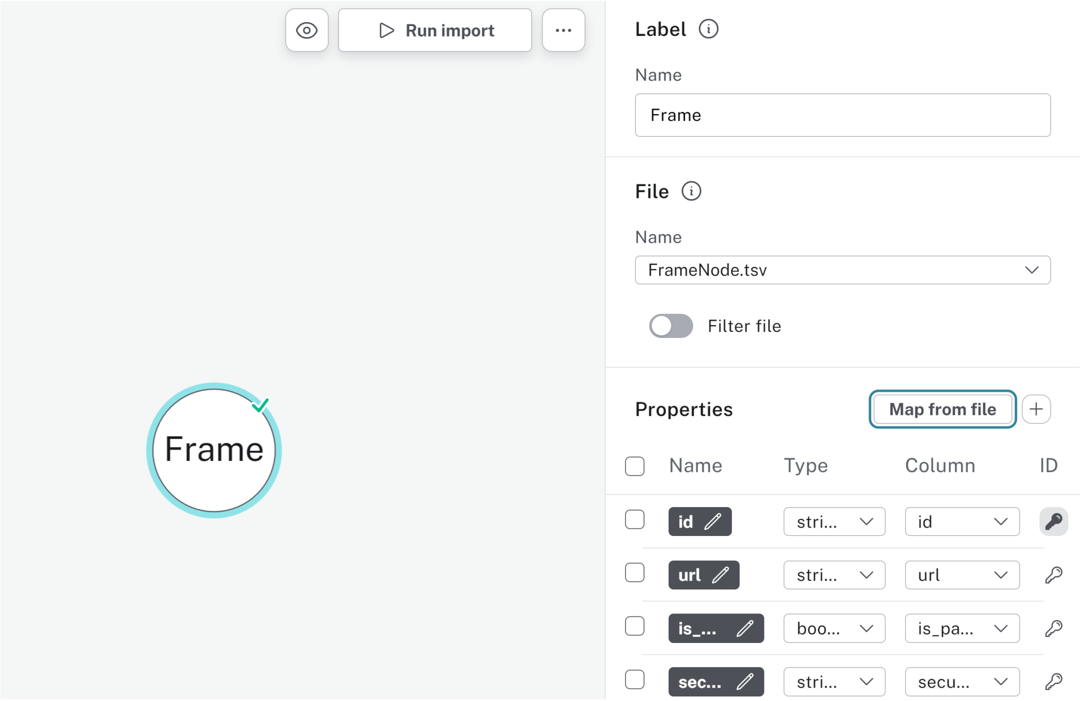
9. Then click on `Map from file` and `Select all` to import the node’s properties.



Map from file ☑ Select all

☑	id	AEB5DAC31F0F07DA...
☑	url	http://cs6264.gtisc.ga...
☑	is_page	True
☑	security_orig...	http://cs6264.gtisc.ga...
☑	mime_type	text/html
☑	frame_id	AEB5DAC31F0F07DA...
☑	loader_id	9260241BC344B6B42...

10. Then you will see a ✔️ symbol on the node, indicating the node class has successfully been constructed.



Label ⓘ

Name

Frame

File ⓘ

Name

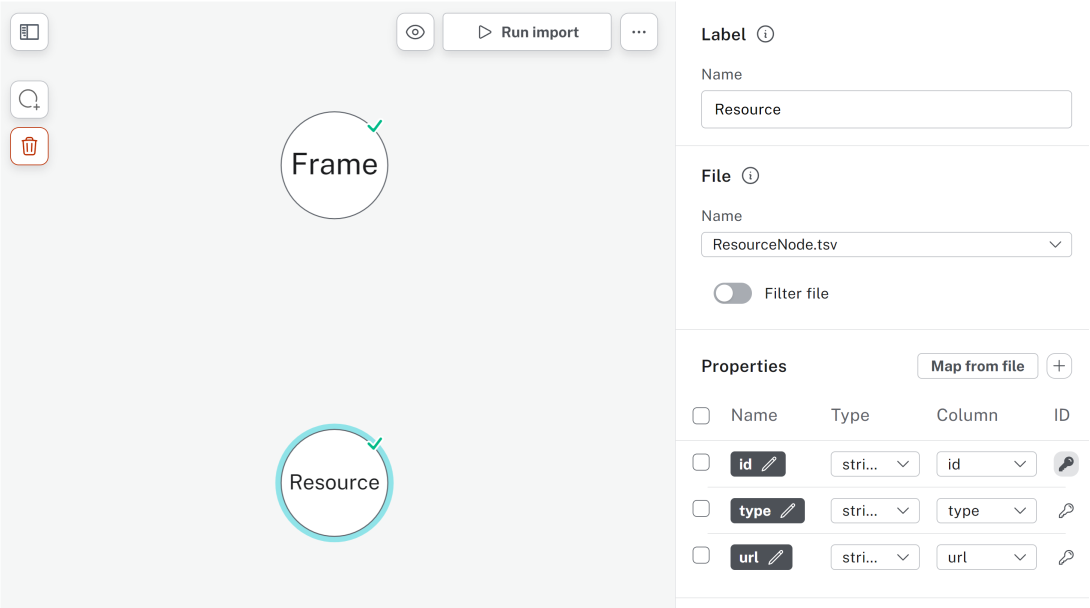
FrameNode.tsv

Filter file

Properties

□	Name	Type	Column	ID
□	id	stri...	id	🔑
□	url	stri...	url	
□	is_...	boo...	is_pa...	
□	sec...	stri...	secu...	

11. Repeat the same procedure to create another node class (e.g., Resource).



Label ⓘ

Name

Resource

File ⓘ

Name

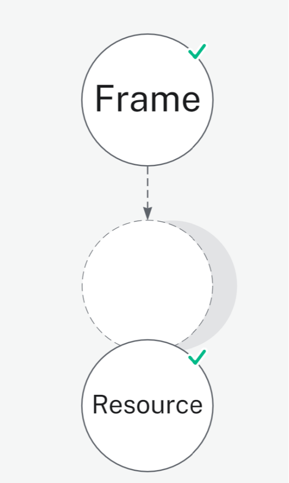
ResourceNode.tsv

Filter file

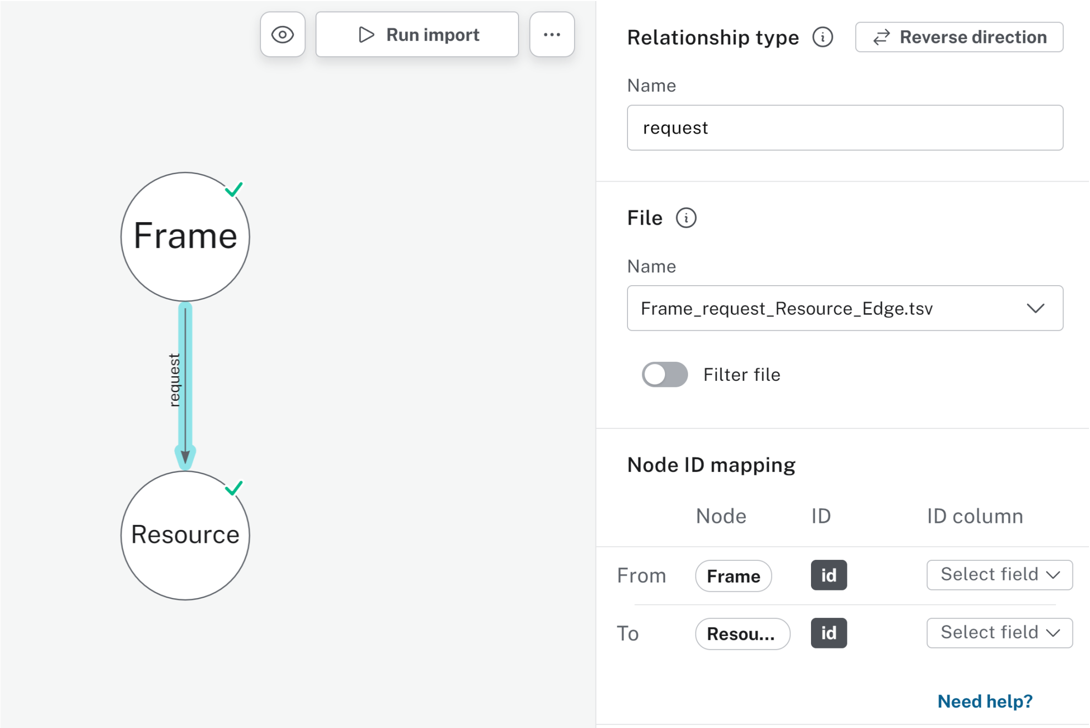
Properties

□	Name	Type	Column	ID
□	id	stri...	id	🔑
□	type	stri...	type	
□	url	stri...	url	

12. Now it’s time to add edges! Move your mouse cursor on the edge of the Frame node and drag a line to the Resource node.



13. Set the relationship name to be `request` and select the corresponding edge log file. **Note**: please keep relationship names all lowercase so that we can differentiate them from node names.



Relationship type

Name

request

File

Name

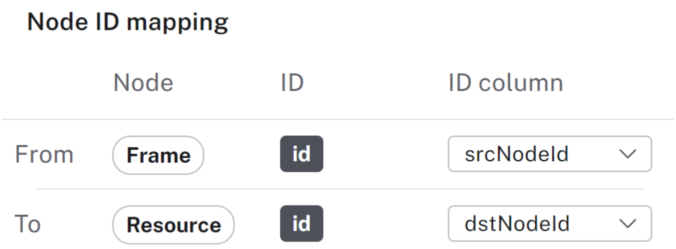
Frame_request_Resource_Edge.tsv

Filter file

Node ID mapping

	Node	ID	ID column
From	Frame	id	Select field
To	Resou...	id	Select field

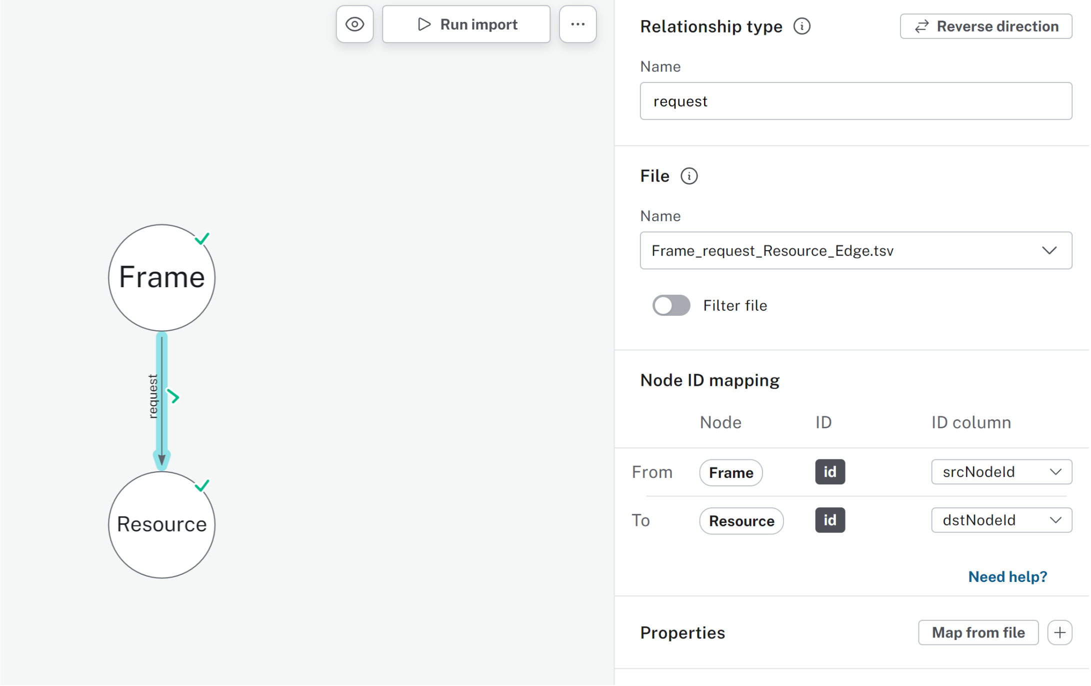
14. Under `Node ID mapping`, select `SrcNodeId` for Frame (the source node) and `dstNodeId` for Resource node (the destination node).



Node ID mapping

	Node	ID	ID column
From	Frame	id	srcNodeId
To	Resource	id	dstNodeId

15. A complete relationship now is constructed (see the three ✔️ symbols). Then click on `Run import`.



Relationship type ⓘ

Reverse direction

Name: request

File ⓘ

Name: Frame_request_Resource_Edge.tsv

○ Filter file

Node ID mapping

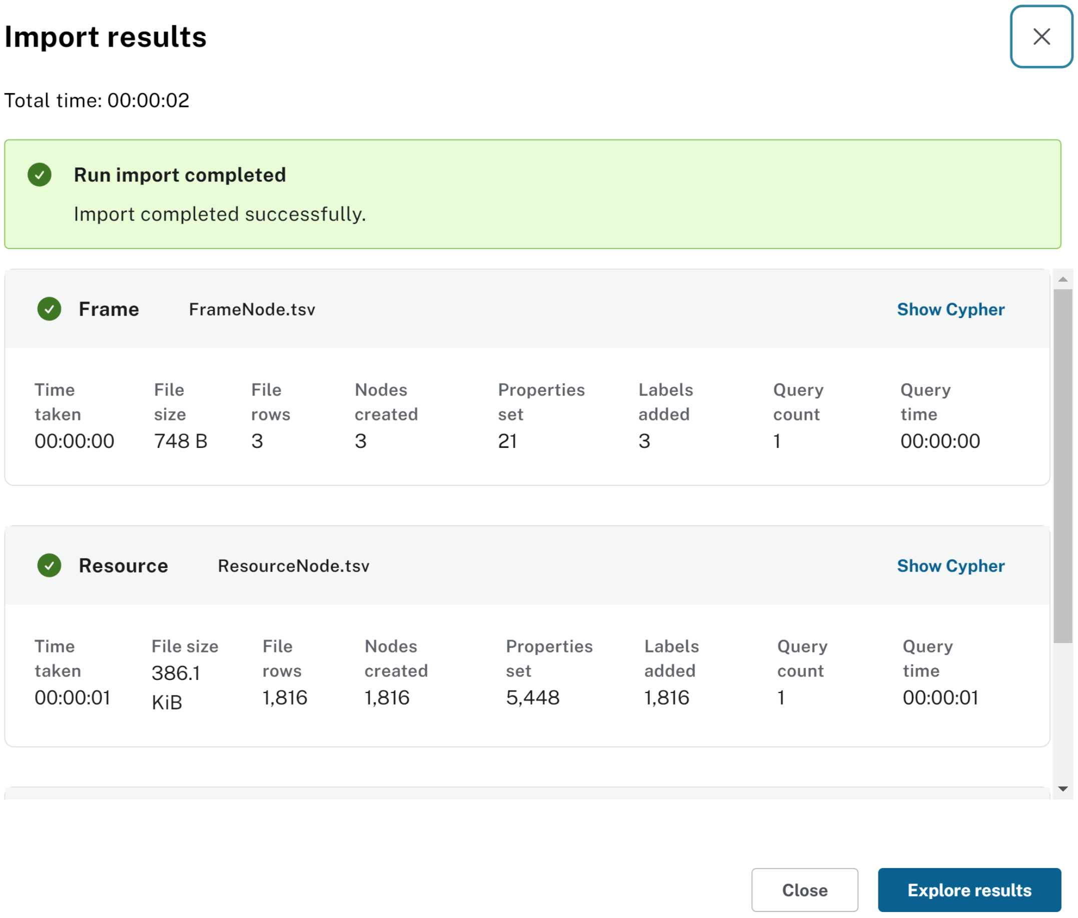
	Node	ID	ID column
From	Frame	id	srcNodeId
To	Resource	id	dstNodeId

Need help?

Properties

Map from file +

16. Click on `Explore results`



Import results

Total time: 00:00:02

✓ Run import completed

Import completed successfully.

✓ Frame FrameNode.tsv

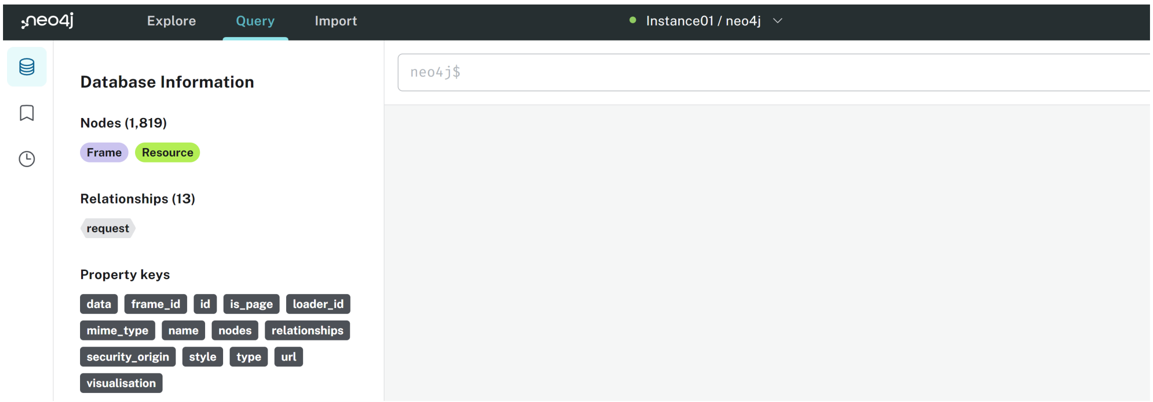
Time taken	File size	File rows	Nodes created	Properties set	Labels added	Query count	Query time
00:00:00	748 B	3	3	21	3	1	00:00:00

✓ Resource ResourceNode.tsv

Time taken	File size	File rows	Nodes created	Properties set	Labels added	Query count	Query time
00:00:01	386.1 KiB	1,816	1,816	5,448	1,816	1	00:00:01

Query

1. Once the graph is constructed. Go to `Query` tab and try your queries!



Database Information

Nodes (1,819)

Frame Resource

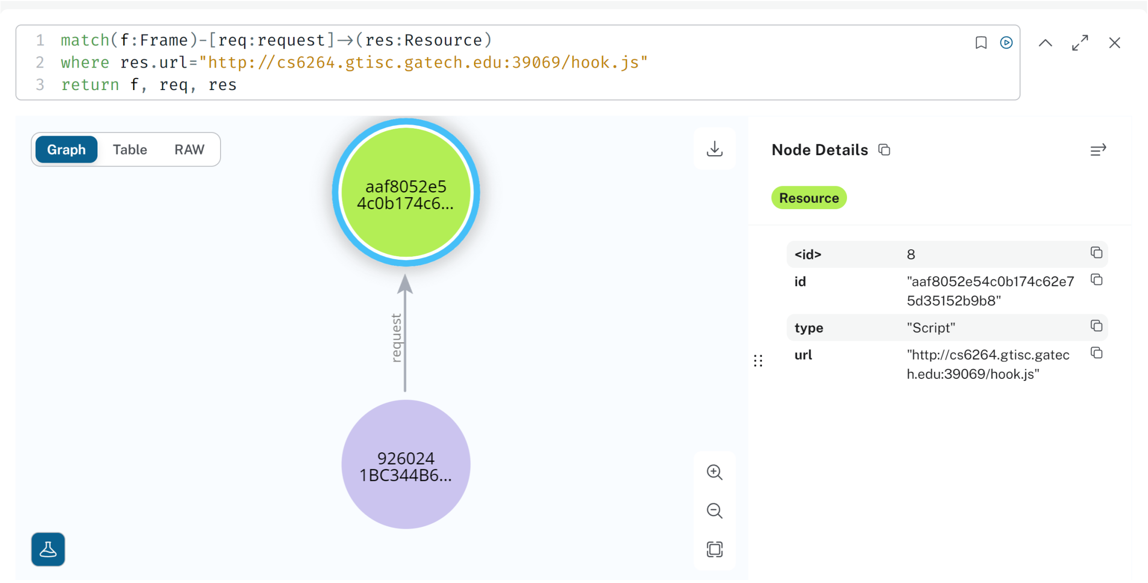
Relationships (13)

request

Property keys

data frame_id id is_page loader_id mime_type name nodes relationships security_origin style type url visualisation

2. Here is just an example to give you a feeling of what cypher (neo4j's query language) looks like. You can find many good online tutorials. (Here are the ones I found helpful video, official doc).

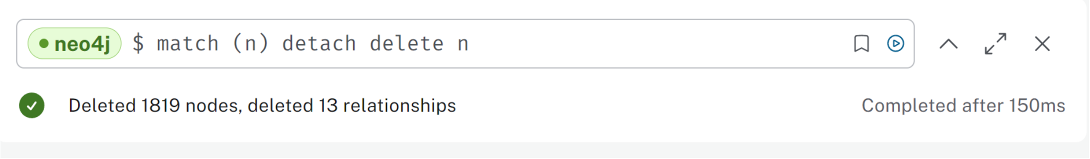


```
match(f:Frame)-[req:request]→(res:Resource)
where res.url="http://cs6264.gtisc.gatech.edu:39069/hook.js"
return f, req, res
```

<id>	8
id	"aaf8052e54c0b174c62e75d35152b9b8"
type	"Script"
url	"http://cs6264.gtisc.gatech.edu:39069/hook.js"

Note: When you want to analyze new logs, you can clear the loaded ones with 2 steps.

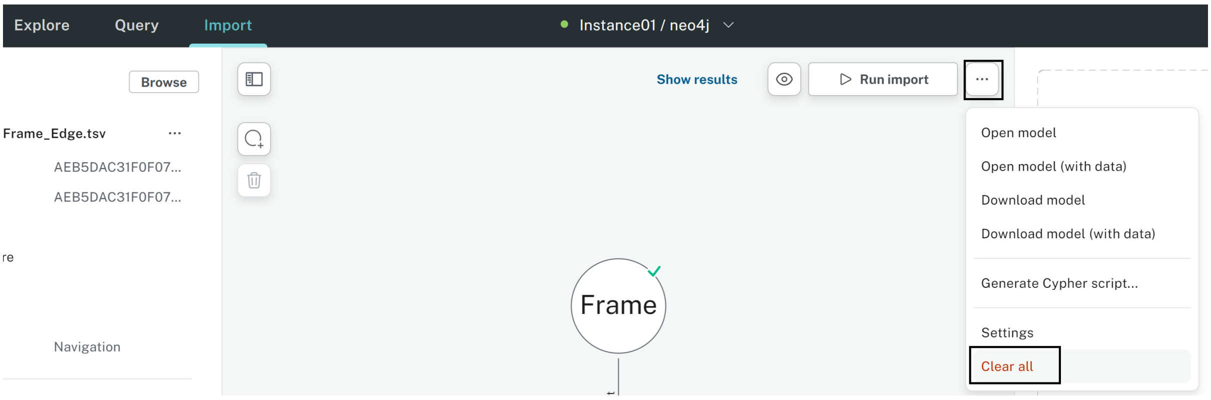
1. Run `match (n) detach delete n` query to delete all the nodes and relationships.



```
neo4j $ match (n) detach delete n
```

```
Deleted 1819 nodes, deleted 13 relationships                    Completed after 150ms
```

2. Go back to `Import`, under `…` next to `Run import`, click on `Clear all`. Then you can start again! (No need to worry about the view in `Explore`.)



Explore
Query
Import
Instance01 / neo4j
Browse
Frame_Edge.tsv
AEB5DAC31F0F07...
AEB5DAC31F0F07...
re
Navigation
Show results
Run import
Open model
Open model (with data)
Download model
Download model (with data)
Generate Cypher script...
Settings
Clear all
Frame