

# Grid

Want to run tests in parallel across multiple machines? Then, Grid is for you.

- 1: [Getting started with Selenium Grid](#)
- 2: [When to Use Grid](#)
- 3: [Selenium Grid Components](#)
- 4: [Configuration of Components](#)
  - 4.1: [Configuration help](#)
  - 4.2: [CLI options in the Selenium Grid](#)
  - 4.3: [TOML configuration options](#)
- 5: [Grid architecture](#)
- 6: [Advanced features of Selenium](#)
  - 6.1: [Observability in Selenium Grid](#)
  - 6.2: [GraphQL query support](#)
  - 6.3: [Grid endpoints](#)

Selenium Grid allows the execution of WebDriver scripts on remote machines by routing commands sent by the client to remote browser instances.

Grid aims to:

- Provide an easy way to run tests in parallel on multiple machines
- Allow testing on different browser versions
- Enable cross platform testing

Interested? Go through the following sections to understand how Grid works, and how to set up your own.

# 1 - Getting started with Selenium Grid

Instructions for a simple Selenium Grid

## Quick start

1. Prerequisites
  - Java 11 or higher installed
  - Browser(s) installed
  - Browser driver(s) [installed and on the PATH](#)
  - Download the Selenium Server jar file from the [latest release](#)
2. Start the Grid
  - `java -jar selenium-server-<version>.jar standalone`
3. Point\* your WebDriver tests to <http://localhost:4444>
4. (Optional) Check running tests and available capabilities by opening your browser at <http://localhost:4444>

\*Wondering how to point your tests to <http://localhost:4444>? Check the [RemoteWebDriver section](#).

To learn more about the different configuration options, go through the sections below.

## Grid roles

Grid is composed by six different [components](#), which gives you the option to deploy it in different ways.

Depending on your needs, you can start each one of them on its own (Distributed), group them in Hub & Node, or all in one on a single machine (Standalone).

## Standalone

**Standalone** combines all Grid [components](#) seamlessly into one. Running a Grid in **Standalone** mode gives you a fully functional Grid with a single command, within a single process. **Standalone** can only run on a single machine.

**Standalone** is also the easiest mode to spin up a Selenium Grid. By default, the server will listen for RemoteWebDriver requests on <http://localhost:4444>. By default, the server will detect the available drivers that it can use from the System [PATH](#).

```
java -jar selenium-server-<version>.jar standalone
```

After starting successfully the Grid in Standalone mode, point your WebDriver tests to <http://localhost:4444>.

Common use cases for **Standalone** are:

- Develop or debug tests using RemoteWebDriver locally
- Running quick test suites before pushing code
- Have a easy to setup Grid in a CI/CD tool (GitHub Actions, Jenkins, etc...)

## Hub and Node

**Hub and Node** is the most used role because it allows to:

- Combine different machines in a single Grid
  - Machines with different operating systems and/or browser versions, for example
- Have a single entry point to run WebDriver tests in different environments
- Scaling capacity up or down without tearing down the Grid

## Hub

A Hub is composed by the following [components](#): Router, Distributor, Session Map, New Session Queue, and Event Bus.

```
java -jar selenium-server-<version>.jar hub
```

By default, the server will listen for `RemoteWebDriver` requests on <http://localhost:4444>.

## Node

During startup time, the **Node** will detect the available drivers that it can use from the System [PATH](#).

The command below assumes the **Node** is running on the same machine where the **Hub** is running.

```
java -jar selenium-server-<version>.jar node
```

### More than one Node on the same machine

#### Node 1

```
java -jar selenium-server-<version>.jar node --port 5555
```

#### Node 2

```
java -jar selenium-server-<version>.jar node --port 6666
```

### Node and Hub on different machines

**Hub** and **Nodes** talk to each other via HTTP and the [Event Bus](#) (the **Event Bus** lives inside the **Hub**). A **Node** sends a message to the **Hub** via the **Event Bus** to start the registration process. When the **Hub** receives the message, reaches out to the **Node** via HTTP to confirm its existence.

To successfully register a **Node** to a **Hub**, it is important to expose the **Event Bus** ports (4442 and 4443 by default) on the **Hub** machine. This also applies for the **Node** port. With that, both **Hub** and **Node** will be able to communicate.

If the **Hub** is using the default ports, the `--hub` flag can be used to register the **Node**

```
java -jar selenium-server-<version>.jar node --hub http://<hub-ip>:444
```

When the **Hub** is not using the default ports, the `--publish-events` and `--subscribe-events` flags are needed.

For example, if the **Hub** uses ports 8886 , 8887 , and 8888

```
java -jar selenium-server-<version>.jar hub --publish-events tcp://<hu
```

The **Node** needs to use those ports to register successfully

```
java -jar selenium-server-<version>.jar node --publish-events tcp://<h
```

# Distributed

When using a Distributed Grid, each component is started separately, and ideally on different machines.

It is important to expose all ports properly in order to allow fluent communication between all components.

- 1. **Event Bus**: enables internal communication between different Grid components.

Default ports are: 4442 , 4443 , and 5557 .

```
java -jar selenium-server-<version>.jar event-bus --publish-events tcp://<event-bus>
```

- 2. **New Session Queue**: adds new session requests to a queue, which will be queried by the Distributor

Default port is 5559 .

```
java -jar selenium-server-<version>.jar sessionqueue --port 5559
```

- 3. **Session Map**: maps session IDs to the **Node** where the session is running

Default **Session Map** port is 5556 . **Session Map** interacts with the **Event Bus**.

```
java -jar selenium-server-<version>.jar sessions --publish-events tcp://<event-bus>
```

- 4. **Distributor**: queries the **New Session Queue** for new session requests, and assigns them to a **Node** when the capabilities match. **Nodes** register to the **Distributor** the way they register to the **Hub** in a **Hub/Node** Grid.

Default **Distributor** port is 5553 . **Distributor** interacts with **New Session Queue**, **Session Map**, **Event Bus**, and the **Node(s)**.

```
java -jar selenium-server-<version>.jar distributor --publish-events tcp://<event-bus>
```

- 5. **Router**: redirects new session requests to the queue, and redirects running sessions requests to the **Node** running that session.

Default **Router** port is 4444 . **Router** interacts with **New Session Queue**, **Session Map**, and **Distributor**.

```
java -jar selenium-server-<version>.jar router --sessions http://<session-queue>
```

- 6. **Node(s)**

Default **Node** port is 5555 .

```
java -jar selenium-server-<version>.jar node --publish-events tcp://<event-bus>
```

# Metadata in tests

Add metadata to your tests and consume it via [GraphQL](#) or visualize parts of it (like `se:name`) through the Selenium Grid UI.

Metadata can be added by prefixing a capability with `se:` . Here is a quick example in Java showing that.

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setCapability("browserVersion", "100");
chromeOptions.setCapability("platformName", "Windows");
// Showing a test name instead of the session id in the Grid UI
chromeOptions.setCapability("se:name", "My simple test");
// Other type of metadata can be seen in the Grid UI by clicking on the
// session info or via GraphQL
chromeOptions.setCapability("se:sampleMetadata", "Sample metadata value");
WebDriver driver = new RemoteWebDriver(new URL("http://gridUrl:4444"),
driver.get("http://www.google.com");
driver.quit();
```

## Querying Selenium Grid

After starting a Grid, there are mainly two ways of querying its status, through the Grid UI or via an API call.

The Grid UI can be reached by opening your preferred browser and heading to <http://localhost:4444>.

API calls can be done through the <http://localhost:4444/status> endpoint or using [GraphQL](#)

For simplicity, all command examples shown in this page assume that components are running locally. More detailed examples and usages can be found in the [Configuring Components](#) section.

## Grid sizes

Choosing a Grid role depends on what operating systems and browsers need to be supported, how many parallel sessions need to be executed, the amount of available machines, and how powerful (CPU, RAM) those machines are.

Creating sessions concurrently relies on the available processors to the **Distributor**. For example, if a machine has 4 CPUs, the **Distributor** will only be able to create up to 4 sessions concurrently.

By default, the maximum amount of concurrent sessions a **Node** supports is limited by the number of CPUs available. For example, if the **Node** machine has 8 CPUs, it can run up to 8 concurrent browser sessions (with the exception of Safari, which is always one). Additionally, it is expected that each browser session should use around 1GB RAM.

In general, it is recommended to have **Nodes** as small as possible. Instead of having a machine with 32 CPUs and 32GB RAM to run 32 concurrent browser sessions, it is better to have 32 small **Nodes** in order to better isolate processes. With this, if a **Node** fails, it will do it in an isolated way. Docker is a good tool to achieve this approach.

Note that the default values (1CPU/1GB RAM per browser) are a recommendation and they could not apply to your context. It is recommended to use them as a reference, but measuring performance continuously will help to determine the ideal values for your environment.

Grid sizes are relative to the amount of supported concurrent sessions and amount of **Nodes**, and there is no “one size fits all”. Sizes mentioned below are rough estimations that can vary between different environments. For example a **Hub/Node** with 120 **Nodes** might work well when the **Hub** has enough resources. Values below are not set on stone, and feedback is welcomed!

## Small

**Standalone** or **Hub/Node** with 5 or less **Nodes**.

## Middle

**Hub/Node** between 6 and 60 **Nodes**.

## Large

**Hub/Node** between 60 and 100 **Nodes**. **Distributed** with over 100 **Nodes**.

## Warning

Selenium Grid must be protected from external access using appropriate firewall permissions.

Failure to protect your Grid could result in one or more of the following occurring:

- You provide open access to your Grid infrastructure
- You allow third parties to access internal web applications and files
- You allow third parties to run custom binaries

See this blog post on [Detectify](#), which gives a good overview of how a publicly exposed Grid could be misused: [Don't Leave your Grid Wide Open](#)

## Further reading

- [Components](#): learn how Grid's internal components relate to each other.
- [Configuration](#): customize your Grid setup.
- [Architecture](#): understand key concepts in Grid.
- [Advanced Features](#): explore more possibilities through Grid's features.

## 2 - When to Use Grid

Is Grid right for you?

When would you use a Selenium Grid?

- To run your tests in parallel, against different browser types, browser versions, operating systems
- To reduce the time needed to execute a test suite

Selenium Grid runs test suites in parallel against multiple machines (called Nodes). For large and long-running test suites, this can save minutes, hours, or perhaps days. This shortens the turnaround time for test results as your application under test (AUT) changes.

Grid can run tests (in parallel) against multiple different browsers, and it can run against multiple instances of the same browser. As an example, let’s imagine a Grid with six Nodes. The first machine has Firefox’s latest version, the second has Firefox “latest minus one”, the third gets the latest Chrome, and the remaining three machines are Mac Minis, which allows for three tests to run in parallel on the latest version of Safari.

Execution time can be expressed as a simple formula:

Number of Tests \* Average Test Time / Number of Nodes = Total Execution Time

15	*	45s	/	1	=	11m 15s	// Without Grid
15	*	45s	/	5	=	2m 15s	// Grid with 5 Nodes
15	*	45s	/	15	=	45s	// Grid with 15 Nodes
100	*	120s	/	15	=	13m 20s	// Would take over 13 minutes

As the test suite is executing, the Grid allocates the tests to run against these browsers as configured in the tests.

A configuration such as this can greatly speed up the execution time of even the largest Selenium test suites.

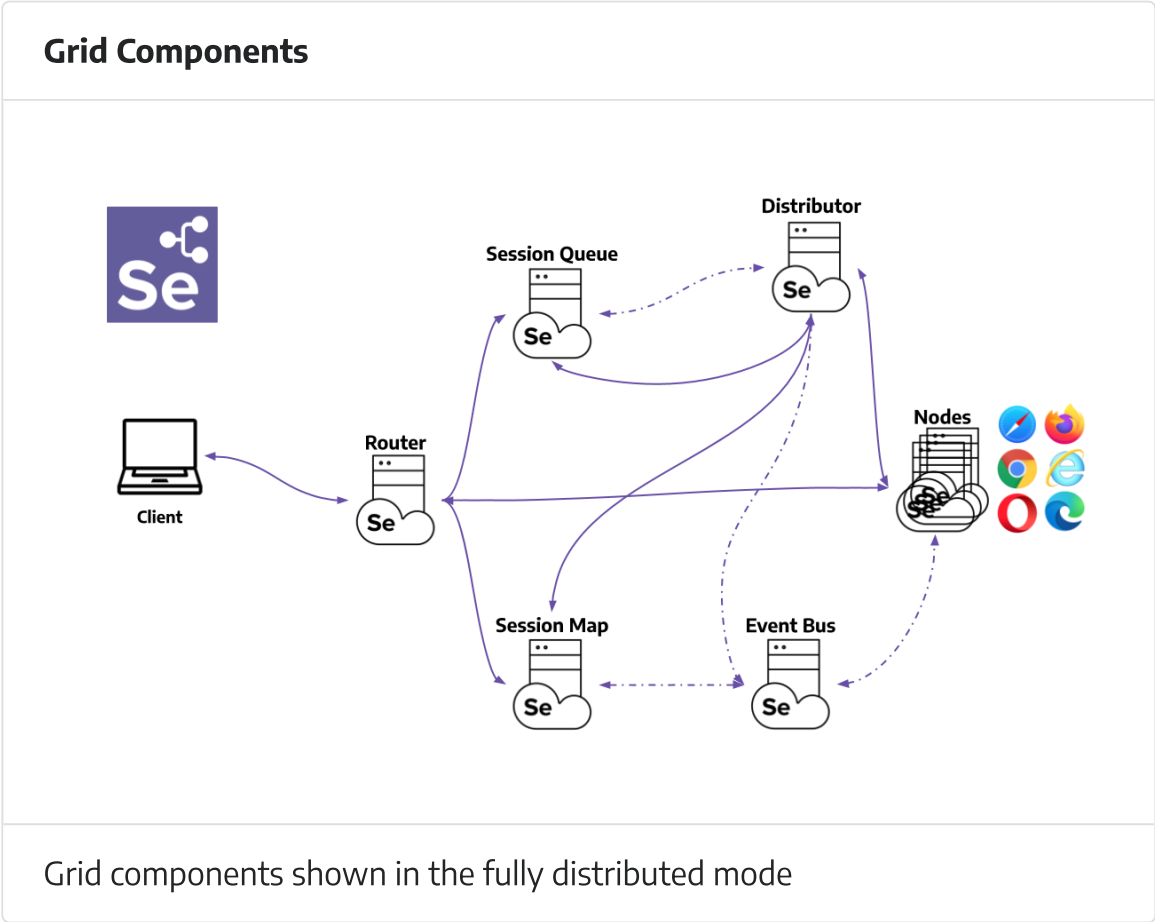
Selenium Grid is a completely native part of the Selenium project, and is maintained in parallel by the same team of committers who work in the core Selenium development. Recognizing the importance of test execution speed, Grid has been a critical part of the Selenium project since the earliest days.



# 3 - Selenium Grid Components

Understand how to use the different Grid components

Selenium Grid 4 is a ground-up rewrite from previous versions. In addition to a comprehensive set of improvements to performance and standards compliance, the different functions of the grid were broken out to reflect a more modern age of computing and software development. Purpose-build for containerization and cloud-distributed scalability, Selenium Grid 4 is a wholly new solution for the modern era.



## Router

The **Router** is the entry point of the Grid, receiving all external requests, and forwards them to the correct component.

If the **Router** receives a new session request, it will be forwarded to the **New Session Queue**.

If the request belongs to an existing session, the **Router** will query the **Session Map** to get the **Node** ID where the session is running, and then the request will be forwarded directly to the **Node**.

The **Router** balances the load in the Grid by sending the requests to the component that is able to handle them better, without overloading any component that is not needed in the process.

## Distributor

The **Distributor** has two main responsibilities:

### Register and keep track of all Nodes and their capabilities

A **Node** registers to the **Distributor** by sending a **Node** registration event through the **Event Bus**. The **Distributor** reads it, and then tries to reach the **Node** via HTTP to confirm its existence. If the request is successfull, the **Distributor** registers the Node and keeps track of all **Nodes** capabilities through the **GridModel**.

### Query the New Session Queue and process any pending new session requests

When a new session request is sent to the **Router**, it gets forwarded to the **New Session Queue**, where it will wait in the queue. The **Distributor** will poll the **New Session Queue** for pending new session requests, and then finds a suitable **Node** where the session can be created. After the session



has been created, the **Distributor** stores in the **Session Map** the relation between the session id and **Node** where the session is being executed.

## Session Map

The **Session Map** is a data store that keeps the relationship between the session id and the **Node** where the session is running. It supports the **Router** in the process of forwarding a request to the **Node**. The **Router** will ask the **Session Map** for the **Node** associated to a session id.

## New Session Queue

The **New Session Queue** holds all the new session requests in a FIFO order. It has configurable parameters for setting the request timeout and request retry interval (how often the timeout will be checked).

The **Router** adds the new session request to the **New Session Queue** and waits for the response. The **New Session Queue** regularly checks if any request in the queue has timed out, if so the request is rejected and removed immediately.

The **Distributor** regularly checks if a slot is available. If so, the **Distributor** polls the **New Session Queue** for the first matching request. The **Distributor** then attempts to create a new session.

Once the requested capabilities match the capabilities of any of the free **Node** slots, the **Distributor** attempts to get the available slot. If all the slots are busy, the **Distributor** will send the request back to the queue. If request times out while retrying or adding to the front of the queue, it will be rejected.

After a session is created successfully, the **Distributor** sends the session information to the **New Session Queue**, which then gets sent back to the **Router**, and finally to the client.

## Node

A Grid can contain multiple **Nodes**. Each **Node** manages the slots for the available browsers of the machine where it is running.

The **Node** registers itself to the **Distributor** through the **Event Bus**, and its configuration is sent as part of the registration message.

By default, the **Node** auto-registers all browser drivers available on the path of the machine where it runs. It also creates one slot per available CPU for Chromium based browsers and Firefox. For Safari, only one slot is created. Through a specific [configuration](#), it can run sessions in Docker containers or relay commands.

A **Node** only executes the received commands, it does not evaluate, make judgments, or control anything other than the flow of commands and responses. The machines where the **Node** is running does not need to have the same operating system as the other components. For example, A Windows **Node** might have the capability of offering IE Mode on Edge as a browser option, whereas this would not be possible on Linux or Mac, and a Grid can have multiple **Nodes** configured with Windows, Mac, or Linux.

## Event Bus

The **Event Bus** serves as a communication path between the **Nodes**, **Distributor**, **New Session Queue**, and **Session Map**. The Grid does most of its internal communication through messages, avoiding expensive HTTP calls. When starting the Grid in its fully distributed mode, the **Event Bus** is the first component that should be started.

Looking forward to using all these components and run your own Grid? Head to the [“Getting Started”](#) section to understand how to put all these pieces together.

# 4 - Configuration of Components

Here you can see how each Grid component can be configured individually based on common configuration values and component-specific configuration values.

## 4.1 - Configuration help

Get information about all the available options to configure Grid.

The help commands display information based on the current code implementation. Hence, it will provide accurate information in case the documentation is not updated. It is the easiest way to learn about Grid 4 configuration for any new version.

### Info Command

The info command provides detailed docs on the following topics:

- Configuring Selenium
- Security
- Session Map setup
- Tracing

### Config help

Quick config help and overview is provided by running:

```
java -jar selenium-server-<version>.jar info config
```

### Security

To get details on setting up the Grid servers for secure communication and node registration:

```
java -jar selenium-server-<version>.jar info security
```

### Session Map setup

By default, Grid uses a local session map to store session information. Grid supports additional storage options like Redis and JDBC - SQL supported databases. To set up different session storage, use the following command to get setup steps:

```
java -jar selenium-server-<version>.jar info sessionmap
```

### Setting up tracing with OpenTelemetry and Jaeger

By default, tracing is enabled. To export traces and visualize them via Jaeger, use the following command for instructions:

```
java -jar selenium-server-<version>.jar info tracing
```

## List the Selenium Grid commands

```
java -jar selenium-server-<version>.jar --config-help
```

It will show all the available commands and description for each one.

## Component help commands

Pass `--help` config option after the Selenium role to get component-specific config information.

### Standalone

```
java -jar selenium-server-<version>.jar standalone --help
```

### Hub

```
java -jar selenium-server-<version>.jar hub --help
```

### Sessions

```
java -jar selenium-server-<version>.jar sessions --help
```

### New Session Queue

```
java -jar selenium-server-<version>.jar sessionqueue --help
```

### Distributor

```
java -jar selenium-server-<version>.jar distributor --help
```

### Router

```
java -jar selenium-server-<version>.jar router --help
```

### Node

```
java -jar selenium-server-<version>.jar node --help
```

# 4.2 - CLI options in the Selenium Grid

All Grid components configuration CLI options in detail.

Different sections are available to configure a Grid. Each section has options can be configured through command line arguments.

A complete description of the component to section mapping can be seen below.

Note that this documentation could be outdated if an option was modified or added but has not been documented yet. In case you bump into this situation, please check the [“Config help”](#) section and feel free to send us a pull request updating this page.

## Sections

	Standalone	Hub	Node	Distributor	Router	Sessions	SessionQueue
<a href="#">Distributor</a>	✓	✓		✓	✓		
<a href="#">Docker</a>	✓		✓				
<a href="#">Events</a>		✓	✓	✓		✓	✓
<a href="#">Logging</a>	✓	✓	✓	✓	✓	✓	✓
<a href="#">Network</a>	✓	✓			✓		
<a href="#">Node</a>	✓		✓				
<a href="#">Router</a>	✓	✓			✓		
<a href="#">Relay</a>	✓		✓				
<a href="#">Server</a>	✓	✓	✓	✓	✓	✓	✓
<a href="#">SessionQueue</a>	✓	✓		✓	✓		✓
<a href="#">Sessions</a>				✓	✓	✓	

## Distributor

Option	Type	Value/Example
<code>--healthcheck-interval</code>	int	120
<code>--distributor</code>	uri	http://localhost:5553

Option	Type	Value/Example
--distributor-host	string	localhost
--distributor-implementation	string	org.openqa.selenium.grid.distributor.local.LocalDistributor
--distributor-port	int	5553
--reject-unsupported-caps	boolean	false
--slot-matcher	string	org.openqa.selenium.grid.data.DefaultSlotMatcher
--slot-selector	string	org.openqa.selenium.grid.distributor.selector.DefaultSlotSelector

## Docker

Option	Type	Value/Example	Description
--------	------	---------------	-------------





Option	Type	Value/Example	Description
--bind-bus	boolean	false	Whether to connect to the Event Bus using a ZeroMQ string. If true, the Event Bus will be bound to the specified string. When false, the Event Bus will be bound to the default string. When true, the Event Bus will be bound to the specified string. When false, the Event Bus will be bound to the default string. When true, the Event Bus will be bound to the specified string. When false, the Event Bus will be bound to the default string.
--events-implementation	string	org.openqa.selenium.events.zeromq.ZeroMqEventBus	Full class name of the non-default event bus implementation.
--publish-events	string	tcp://*:4442	Connection string for publishing events to the Event Bus.
--subscribe-events	string	tcp://*:4443	Connection string for subscribing to events from the Event Bus.

Logging

Option	Type	Value/Example	Description
--http-logs	boolean	false	Enable http logging. Tracing : http://localhost:4444/
--log-encoding	string	UTF-8	Log encoding
--log	string	Windows path example : '\path\to\file\gridlog.log' or 'C:\path\path\to\file\gridlog.log'  Linux/Unix/MacOS path example : '/path/to/file/gridlog.log'	File to write out logs. Ensure the path is valid for the operating system's file path.

Option	Type	Value/Example	Description
--log-level	string	"INFO"	Log level. Default logging level here <a href="https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html">https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html</a>
--plain-logs	boolean	true	Use plain log lines
--structured-logs	boolean	false	Use structured logs
--tracing	boolean	true	Enable trace collection
--log-timestamp-format	string	HH:mm:ss.SSS	Allows the configure log time

Network

Option	Type	Value/Example	Description
--relax-checks	boolean	false	Relax checks on origin header and content type of incoming requests, in contravention of strict W3C spec compliance.

Node

Option	Type	Value/Example	Description
--detect-drivers	boolean	true	Automatically detect drivers
--driver-configuration	string[]	display-name="Firefox Nightly" max-sessions=2 webdriver-path="/usr/local/bin/geckodriver" stereotype='{ "browserName": "firefox", "browserVersion": "86", "moz:firefoxOptions": { "binary": "/Applications/Firefox Nightly.app/Contents/MacOS/firefox-bin" } }'	Load a remote WebDriver instance
--driver-factory	string[]	org.openqa.selenium.example.LynxDriverFactory '{ "browserName": "lynx" }'	Use a custom driver factory
--driver-implementation	string[]	"firefox"	Default driver implementation

Option	Type	Value/Example	Description
--node-implementation	string	"org.openqa.selenium.grid.node.local.LocalNodeFactory"	Factory class used to create nodes
--grid-url	string	https://grid.example.com	URL of the Selenium Grid
--heartbeat-period	int	60	Heartbeat period in seconds
--max-sessions	int	8	Maximum number of sessions per node
--override-max-sessions	boolean	false	Override the maximum number of sessions per node
--register-cycle	int	10	Register cycle in seconds
--register-period	int	120	Register period in seconds
--session-timeout	int	300	Session timeout in seconds

Option	Type	Value/Example	Description
--vnc-env-var	string	START_XVFB	Environment variable to start the VNC session.
--no-vnc-port	int	7900	If the VNC port is not specified, the default is 7900.
--drain-after-session-count	int	1	Drain the VNC session after the specified number of sessions.
--hub	string	http://localhost:4444	URL of the Selenium Hub.
--enable-cdp	boolean	true	Enable Chrome DevTools Protocol (CDP) for the browser.

Relay

Option	Type	Value/Example	Description
--service-url	string	http://localhost:4723	URL for connecting to the service that supports WebDriver commands like an Appium server or a cloud service.
--service-host	string	localhost	Host name where the service that supports WebDriver commands is running
--service-port	int	4723	Port where the service that supports WebDriver commands is running
--service-status-endpoint	string	/status	Optional, endpoint to query the WebDriver service status, an HTTP 200 response is expected
--service-configuration	string[]	max-sessions=2 stereotype='{ "browserName": "safari", "platformName": "iOS", "appium:platformVersion": "14.5"}'	Configuration for the service where calls will be relayed to. It is recommended to provide this type of configuration through a toml config file to improve readability.

Router

Option	Type	Value/Example	Description
--password	string	myStrongPassword	Password clients must use to connect to the server. Both this and the username need to be set in order to be used.
--username	string	admin	User name clients must use to connect to the server. Both this and the password need to be set in order to be used.

Server

Option	Type	Value/Example	Description
--allow-cors	boolean	true	Whether the Selenium server should allow web browser connections from any host
--host	string	localhost	Server IP or hostname: usually determined automatically.
--bind-host	boolean	true	Whether the server should bind to the host address/name, or only use it to report its reachable url. Helpful in complex network topologies where the server cannot report itself with the current IP/hostname but rather an external IP or hostname (e.g. inside a Docker container)

Option	Type	Value/Example	Description
--https-certificate	path	/path/to/cert.pem	Server certificate for https. Get more detailed information by running “java -jar selenium-server.jar info security”
--https-private-key	path	/path/to/key.pkcs8	Private key for https. Get more detailed information by running “java -jar selenium-server.jar info security”
--max-threads	int	24	Maximum number of listener threads. Default value is: (available processors) * 3.
--port	int	4444	Port to listen on. There is no default as this parameter is used by different components, for example, Router/Hub/Standalone will use 4444 and Node will use 5555.

## SessionQueue

Option	Type	Value/Example	Description
--sessionqueue	uri	http://localhost:1237	Address of the session queue server.
-sessionqueue-host	string	localhost	Host on which the session queue server is listening.
--sessionqueue-port	int	1234	Port on which the session queue server is listening.
--session-request-timeout	int	300	Timeout in seconds. A new incoming session request is added to the queue. Requests sitting in the queue for longer than the configured time will timeout.
--session-retry-interval	int	5	Retry interval in seconds. If all slots are busy, new session request will be retried after the given interval.

## Sessions

Option	Type	Value/Example	Description
--sessions	uri	http://localhost:1234	Address of the session map server.
--sessions-host	string	localhost	Host on which the session map server is listening.
--sessions-port	int	1234	Port on which the session map server is listening.

# Configuration examples

All the options mentioned above can be used when starting the Grid components. They are a good way of exploring the Grid options, and trying out values to find a suitable configuration.

We recommend the use of [Toml files](#) to configure a Grid. Configuration files improve readability, and you can also check them in source control.

When needed, you can combine a Toml file configuration with CLI arguments.

## Command-line flags

To pass config options as command-line flags, identify the valid options for the component and follow the template below.

```
java -jar selenium-server-<version>.jar <component> --<option> value
```

## Standalone, setting max sessions and main port

```
java -jar selenium-server-<version>.jar standalone --max-sessions 4 --port 4444
```

## Hub, setting a new session request timeout, a main port, and disabling tracing

```
java -jar selenium-server-<version>.jar hub --session-request-timeout 500 --port 3333
```

## Node, with 4 max sessions, with debug(fine) log, 7777 as port, and only with Firefox and Edge

```
java -jar selenium-server-<version>.jar node --max-sessions 4 --log-level "fine" --port 7777 --capabilities '{"browserName":"firefox","browserVersion":"78.0","platform":"linux"}
```

## Distributor, setting Session Map server url, Session Queue server url, and disabling bus

```
java -jar selenium-server-<version>.jar distributor --sessions http://localhost:5556
```

## Setting custom capabilities for matching specific Nodes

**Important:** Custom capabilities need to be set in the configuration in all Nodes. They also need to be included always in every session request.

### Start the Hub

```
java -jar selenium-server-<version>.jar hub
```

### Start the Node A with custom cap set to **true**

```
java -jar selenium-server-<version>.jar node --detect-drivers false --driver-config
```



Start the Node B with custom cap set to **false**

```
java -jar selenium-server-<version>.jar node --detect-drivers false --driver-configu
```

Matching Node A

```
ChromeOptions options = new ChromeOptions();  
options.setCapability("gsg:customcap", true);  
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444")  
driver.get("https://selenium.dev");  
driver.quit();
```

Set the custom capability to **false** in order to match the Node B.

## 4.3 - TOML configuration options

Grid configuration examples using Toml files.

All the options shown in [CLI options](#) can be configured through a [TOML](#) file. This page shows configuration examples for the different Grid components.

Note that this documentation could be outdated if an option was modified or added but has not been documented yet. In case you bump into this situation, please check the [“Config help”](#) section and feel free to send us a pull request updating this page.

### Overview

Selenium Grid uses [TOML](#) format for config files. The config file consists of sections and each section has options and its respective value(s).

Refer to the [TOML documentation](#) for detailed usage guidance. In case of parsing errors, validate the config using [TOML linter](#).

The general configuration structure has the following pattern:

```
[section1]
option1="value"

[section2]
option2=["value1", "value2"]
option3=true
```

Below are some examples of Grid components configured with a Toml file, the component can be started in the following way:

```
java -jar selenium-server-<version>.jar <component> --config /path/to/file/<file-name>
```

### Standalone

A Standalone server, running on port 4449, and a new session request timeout of 500 seconds.

```
[server]
port = 4449

[sessionqueue]
session-request-timeout = 500
```

### Specific browsers and a limit of max sessions

A Standalone server or a Node which only has Firefox and Chrome enabled by default.

```
[node]
drivers = ["chrome", "firefox"]
max-sessions = 3
```

## Configuring and customising drivers

Standalone or Node server with customised drivers, which allows things like having Firefox Beta or Nightly, and having different browser versions.

```
[node]
detect-drivers = false
[[node.driver-configuration]]
max-sessions = 100
display-name = "Firefox Nightly"
stereotype = "{\"browserName\": \"firefox\", \"browserVersion\": \"93\"}"
[[node.driver-configuration]]
display-name = "Chrome Beta"
stereotype = "{\"browserName\": \"chrome\", \"browserVersion\": \"94\"}"
[[node.driver-configuration]]
display-name = "Chrome Dev"
stereotype = "{\"browserName\": \"chrome\", \"browserVersion\": \"95\"}"
webdriver-executable = '/path/to/chromedriver/95/chromedriver'
```

## Standalone or Node with Docker

A Standalone or Node server that is able to run each new session in a Docker container. Disabling drivers detection, having maximum 2 concurrent sessions. Stereotypes configured need to be mapped to a Docker image, and the Docker daemon needs to be exposed via http/tcp. In addition, it is possible to define which device files, accessible on the host, will be available in containers through the `devices` property. Refer to the [docker](#) documentation for more information about how docker device mapping works.

```
[node]
detect-drivers = false
max-sessions = 2

[docker]
configs = [
    "selenium/standalone-chrome:93.0", "{\"browserName\": \"chrome\", \"browserVersion\": \"93\"}",
    "selenium/standalone-firefox:92.0", "{\"browserName\": \"firefox\", \"browserVersion\": \"92\"}"
]
#Optionally define all device files that should be mapped to docker container
#devices = [
#    "/dev/kvm:/dev/kvm"
#]
url = "http://localhost:2375"
video-image = "selenium/video:latest"
```

## Relaying commands to a service endpoint that supports WebDriver

It is useful to connect an external service that supports WebDriver to Selenium Grid. An example of such service could be a cloud provider or an Appium server. In this way, Grid can enable more coverage to platforms and versions not present locally.

The following is an example of connecting an Appium server to Grid.

```
[node]
detect-drivers = false

[relay]
# Default Appium/Cloud server endpoint
```

```
url = "http://localhost:4723/wd/hub"
status-endpoint = "/status"
# Stereotypes supported by the service. The initial number is "max-ses
# that many test slots to that particular configuration
configs = [
    "5", {"browserName\: \"chrome\", \"platformName\: \"android\", \"
```

## Basic auth enabled

It is possible to protect a Grid with basic auth by configuring the Router/Hub/Standalone with a username and password. This user/password combination will be needed when loading the Grid UI or starting a new session.

```
[router]
username = "admin"
password = "myStrongPassword"
```

Here is a Java example showing how to start a session using the configured user and password.

```
URL gridUrl = new URL("http://admin:myStrongPassword@localhost:4444");
RemoteWebDriver webDriver = new RemoteWebDriver(gridUrl, new ChromeOpt
```

## Setting custom capabilities for matching specific Nodes

**Important:** Custom capabilities need to be set in the configuration in all Nodes. They also need to be included always in every session request.

```
[node]
detect-drivers = false

[[node.driver-configuration]]
display-name = "firefox"
stereotype = '{"browserName": "firefox", "platformName": "macOS", "bro
max-sessions = 5
```

Here is a Java example showing how to match that Node

```
FirefoxOptions options = new FirefoxOptions();
options.setCapability("networkname:applicationName", "node_1");
options.setCapability("nodename:applicationName", "app_1");
options.setBrowserVersion("96");
options.setPlatformName("macOS");
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444")
driver.get("https://selenium.dev");
driver.quit();
```

# 5 - Grid architecture

The Grid is designed as a set of components that all fulfill a role in maintaining the Grid. It can seem quite complicated, but hopefully this document can help clear up any confusion.

## The Key Components

The main components of the Grid are:

### Event Bus

Used for sending messages which may be received asynchronously between the other components.

### New Session Queue

Maintains a list of incoming sessions which have yet to be assigned to a Node by the Distributor.

### Distributor

Responsible for maintaining a model of the available locations in the Grid where a session may run (known as "slots") and taking any incoming [new session](#) requests and assigning them to a slot.

### Node

Runs a [WebDriver session](#). Each session is assigned to a slot, and each node has one or more slots.

### Session Map

Maintains a mapping between the [session ID](#) and the address of the Node the session is running on.

### Router

Acts as the front-end of the Grid. This is the only part of the Grid which *may* be exposed to the wider Web (though we strongly caution against it). This routes incoming requests to either the New Session Queue or the Node on which the session is running.

While discussing the Grid, there are some other useful concepts to keep in mind:

- A **slot** is the place where a session can run.
- Each slot has a **stereotype**. This is the minimal set of capabilities that a [new session](#) session request must match before the Distributor will send that request to the Node owning the slot.
- The **Grid Model** is how the Distributor tracks the state of the Grid. As the name suggests, this may sometimes fall out of sync with reality (perhaps because the Distributor has only just started). It is used in preference to querying each Node so that the Distributor can quickly assign a slot to a New Session request.

## Synchronous and Asynchronous Calls

There are two main communication mechanisms used within the Grid:

1. Synchronous “REST-ish” JSON over HTTP requests.
2. Asynchronous events sent to the Event Bus.

How do we pick which communication mechanism to use? After all, we could model the entire Grid in an event-based way, and it would work out just fine.


The answer is that if the action being performed is synchronous (eg. most WebDriver calls), or if missing the response would be problematic, the Grid uses a synchronous call. If, instead, we want to broadcast information to anyone who’s interested, or if missing the response doesn’t matter, then we prefer to use the event bus.








One interesting thing to note is that the async calls are more decoupled from their listeners than the synchronous calls are.

## Start Up Sequence and Dependencies Between Components

Although the Grid is designed to allow components to start up in any order, conceptually the order in which components starts is:

1. The Event Bus and Session Map start first. These have no other dependencies, not even on each other, and so are safe to start in parallel.
2. The Session Queue starts next.
3. It is now possible to start the Distributor. This will periodically connect to the Session Queue and poll for jobs, though this polling might be initiated either by an event (that a New Session has been added to the queue) or at regular intervals.
4. The Router(s) can be started. New Session requests will be directed to the Session Queue, and the Distributor will attempt to find a slot to run the session on.
5. We are now able to start a Node. See below for details about how the Node is registered with the Grid. Once registration is complete, the Grid is ready to serve traffic.

You can picture the dependencies between components this way, where a “” indicates that there is a synchronous dependency between the components.

	Event Bus	Distributor	Node	Router	Session Map	Session Queue
Event Bus	X					
Distributor		X				
Node			X			
Router				X		
Session Map					X	
Session Queue						X

## Node Registration

The process of registering a new Node to the Grid is lightweight.

1. When the Node starts, it should emit a “heart beat” event on a regular basis. This heartbeat contains the [node status](#).
2. The Distributor listens for the heart beat events. When it sees one, it attempts to `GET` the `/status` endpoint of the Node. It is from this information that the Grid is set up.

The Distributor will use the same `/status` endpoint to check the Node on a regular basis, but the Node should continue sending heart beat events even after started so that a Distributor without a persistent store of the Grid state can be restarted and will (eventually) be up to date and correct.

## The Node Status Object

The Node Status is a JSON blob with the following fields:

Name	Type	Description
availability	string	A string which is one of <code>up</code> , <code>draining</code> , or <code>down</code> . The important one is <code>draining</code> , which indicates that no new sessions should be sent to the Node, and once the last session on it closes, the Node will exit or restart.
externalUrl	string	The URI that the other components in the Grid should connect to.
lastSessionCreated	integer	The epoch timestamp of when the last session was created on this Node. The Distributor will attempt to send new sessions to the Node that has been idle longest if all other things are equal.

Name	Type	Description
maxSessionCount	integer	Although a session count can be inferred by counting the number of available slots, this integer value is used to determine the maximum number of sessions that should be running simultaneously on the Node before it is considered “full”.
nodeId	string	A UUID used to identify this instance of the Node.
osInfo	object	An object with <code>arch</code> , <code>name</code> , and <code>version</code> fields. This is used by the Grid UI and the GraphQL queries.
slots	array	An array of Slot objects (described below)
version	string	The version of the Node (for Selenium, this will match the Selenium version number)

It is recommended to put values in all fields.

## The Slot Object

The Slot object represents a single slot within a Node. A “slot” is where a single session may be run. It is possible that a Node will have more slots than it can run concurrently. For example, a node may be able to run up 10 sessions, but they could be any combination of Chrome, Edge, or Firefox; in this case, the Node would indicate a “max session count” of 10, and then also say it has 10 slots for Chrome, 10 for Edge, and 10 for Firefox.

Name	Type	Description
id	string	UUID to refer to the slot
lastStarted	string	When the slot last had a session started, in ISO-8601 format
stereotype	object	The minimal set of <a href="#">capabilities</a> this slot will match against. A minimal example is <code>{"browserName": "firefox"}</code>
session	object	The Session object (see below)

## The Session Object

This represents a running session within a slot

Name	Type	Description
capabilities	object	The actual capabilities provided by the session. Will match the return value from the <a href="#">new session</a> command
startTime	string	The start time of the session in ISO-8601 format
stereotype	object	The minimal set of <a href="#">capabilities</a> this slot will match against. A minimal example is <code>{"browserName": "firefox"}</code>
uri	string	The URI used by the Node to communicate with the session



# 6 - Advanced features of Selenium

To get all the details of the advanced features, understand how it works, and how to set up your own, please browse thorough the following sections.

## 6.1 - Observability in Selenium Grid

### Table of Contents

- [Selenium Grid](#)
- [Observability](#)
  - [Distributed tracing](#)
  - [Event logging](#)
- [Grid Observability](#)
  - [Visualizing Traces](#)
  - [Leveraging event logs](#)
- [References](#)

### Selenium Grid

Grid aids in scaling and distributing tests by executing tests on various browser and operating system combinations.

### Observability

Observability has three pillars: traces, metrics and logs. Since Selenium Grid 4 is designed to be fully distributed, observability will make it easier to understand and debug the internals.

### Distributed tracing

A single request or transaction spans multiple services and components. Tracing tracks the request lifecycle as each service executes the request. It is useful in debugging in an error scenario. Some key terms used in tracing context are:

**Trace** Tracing allows one to trace a request through multiple services, starting from its origin to its final destination. This request's journey helps in debugging, monitoring the end-to-end flow, and identifying failures. A trace depicts the end-to-end request flow. Each trace has a unique id as its identifier.

**Span** Each trace is made up of timed operations called spans. A span has a start and end time and it represents operations done by a service. The granularity of span depends on how it is instrumented. Each span has a unique identifier. All spans within a trace have the same trace id.

**Span Attributes** Span attributes are key-value pairs which provide additional information about each span.

**Events** Events are timed-stamped logs within a span. They provide additional context to the existing spans. Events also contain key-value pairs as event attributes.

### Event logging

Logging is essential to debug an application. Logging is often done in a human-readable format. But for machines to search and analyze the logs, it has to have a well-defined format. Structured logging is a common practice of recording logs consistently in a fixed format. It commonly contains fields like:

- Timestamp
- Logging level
- Logger class
- Log message (This is further broken down into fields relevant to the operation where the log was recorded)

Logs and events are closely related. Events encapsulate all the possible information available to do a single unit of work. Logs are essentially subsets of an event. At the crux, both aid in debugging. Refer following resources for detailed understanding:

1. <https://www.honeycomb.io/blog/how-are-structured-logs-different-from-events/>
2. <https://charity.wtf/2019/02/05/logs-vs-structured-events/>

## Grid Observability

Selenium server is instrumented with tracing using OpenTelemetry. Every request to the server is traced from start to end. Each trace consists of a series of spans as a request is executed within the server. Most spans in the Selenium server consist of two events:

1. Normal event - records all information about a unit of work and marks successful completion of the work.
2. Error event - records all information till the error occurs and then records the error information. Marks an exception event.

Running Selenium server

1. [Standalone](#)
2. [Hub and Node](#)
3. [Fully Distributed](#)
4. [Docker](#)

## Visualizing Traces

All spans, events and their respective attributes are part of a trace. Tracing works while running the server in all of the above-mentioned modes.

By default, tracing is enabled in the Selenium server. Selenium server exports the traces via two exporters:

1. Console - Logs all traces and their included spans at FINE level. By default, Selenium server prints logs at INFO level and above. The **log-level** flag can be used to pass a logging level of choice while running the Selenium Grid jar/s.

```
java -jar selenium-server-4.0.0-<selenium-version>.jar standalone --log-level=FINE
```

2. Jaeger UI - OpenTelemetry provides the APIs and SDKs to instrument traces in the code. Whereas Jaeger is a tracing backend, that aids in collecting the tracing telemetry data and providing querying, filtering and visualizing features for the data.

Detailed instructions of visualizing traces using Jaeger UI can be obtained by running the command :

```
java -jar selenium-server-4.0.0-<selenium-version>.jar info tracing
```

[A very good example and scripts to run the server and send traces to Jaeger](#)

## Leveraging event logs

Tracing has to be enabled for event logging as well, even if one does not wish to export traces to visualize them.

**By default, tracing is enabled. No additional parameters need to be passed to see logs on the console.** All events within a span are logged at FINE level. Error events are logged at WARN level.

All event logs have the following fields :

Field	Field value	Description
Event time	eventId	Timestamp of the event record in epoch nanoseconds.
Trace Id	tracedId	Each trace is uniquely identified by a trace id.
Span Id	spanId	Each span within a trace is uniquely identified by a span id.
Span Kind	spanKind	Span kind is a property of span indicating the type of span. It helps in understanding the nature of the unit of work done by the Span.
Event name	eventName	This maps to the log message.
Event attributes	eventAttributes	This forms the crux of the event logs, based on the operation executed, it has JSON formatted key-value pairs. This also includes a handler class attribute, to show the logger class.

Sample log

```
FINE [LoggingOptions$1.lambda$export$1] - {
  "traceId": "fc8aef1d44b3cc8bc09eb8e581c4a8eb",
  "spanId": "b7d3b9865d3ddd45",
  "spanKind": "INTERNAL",
  "eventTime": 1597819675128886121,
  "eventName": "Session request execution complete",
  "attributes": {
    "http.status_code": 200,
    "http.handler_class": "org.openqa.selenium.grid.router.HandleSession",
    "http.url": "\u002fsession\u002fd35257f104bb43fdb06242953f4c85",
    "http.method": "DELETE",
    "session.id": "dd35257f104bb43fdb06242953f4c85"
  }
}
```

In addition to the above fields, based on [OpenTelemetry specification](#) error logs consist of :

Field	Field value	Description
Exception type	exception.type	The class name of the exception.
Exception message	exception.message	Reason for the exception.
Exception stacktrace	exception.stacktrace	Prints the call stack at the point of time when the exception was thrown. Helps in understanding the origin of the exception.

Sample error log

```
WARN [LoggingOptions$1.lambda$export$1] - {
  "traceId": "7efa5ea57e02f89cdf8de586fe09f564",
  "spanId": "914df6bc9a1f6e2b",
  "spanKind": "INTERNAL",
  "eventTime": 1597820253450580272,
  "eventName": "exception",
  "attributes": {
    "exception.type": "org.openqa.selenium.ScriptTimeoutException",
    "exception.message": "Unable to execute request: java.sql.SQLSyntaxErrorException",
    "exception.stacktrace": "org.openqa.selenium.ScriptTimeoutException: java.sql.SQLSyntaxErrorException: Unable to execute request: java.sql.SQLSyntaxErrorException",
    "http.handler_class": "org.openqa.selenium.grid.distributor.remote.RemoteDistributor",
    "http.url": "\u002fsession",
    "http.method": "POST"
  }
}
```

Note: Logs are pretty printed above for readability. Pretty printing for logs is turned off in Selenium server.

The steps above should set you up for seeing traces and logs.

# References

- 1. [Understanding Tracing](#)
- 2. [OpenTelemetry Tracing API Specification](#)
- 3. [Selenium Wiki](#)
- 4. [Structured logs vs events](#)
- 5. [Jaeger framework](#)

# 6.2 - GraphQL query support

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. It gives users the power to ask for exactly what they need and nothing more.

## Enums

Enums represent possible sets of values for a field.

For example, the `node` object has a field called `status` . The state is an enum (specifically, of type `status` ) because it may be `UP` , `DRAINING` or `UNAVAILABLE` .

## Scalars

Scalars are primitive values: `Int` , `Float` , `String` , `Boolean` ,Or `ID` .

When calling the GraphQL API, you must specify nested subfield until you return only scalars.

## Structure of the Schema

The structure of grid schema is as follows:

```
{
  session(id: "<session-id>") : {
    id,
    capabilities,
    startTime,
    uri,
    nodeId,
    nodeUri,
    sessionDurationMillis
    slot : {
      id,
      stereotype,
      lastStarted
    }
  }
  grid: {
    uri,
    totalSlots,
    nodeCount,
    maxSession,
    sessionCount,
    version,
    sessionQueueSize
  }
  sessionsInfo: {
    sessionQueueRequests,
    sessions: [
      {
        id,
        capabilities,
        startTime,
        uri,
        nodeId,
        nodeUri,
        sessionDurationMillis
        slot : {
```

```
        id,
        stereotype,
        lastStarted
      }
    }
  ]
}
nodesInfo: {
  nodes : [
    {
      id,
      uri,
      status,
      maxSession,
      slotCount,
      sessions: [
        {
          id,
          capabilities,
          startTime,
          uri,
          nodeId,
          nodeUri,
          sessionDurationMillis
          slot : {
            id,
            stereotype,
            lastStarted
          }
        }
      ],
      sessionCount,
      stereotypes,
      version,
      osInfo: {
        arch,
        name,
        version
      }
    }
  ]
}
```

# Querying GraphQL

The best way to query GraphQL is by using `curl` requests. The query is interpreted as JSON. Ensure double quotes are properly escaped to avoid unexpected errors. GraphQL allows you to fetch only the data that you want, nothing more nothing less.

Some of the example GraphQL queries are given below. You can build your own queries as you like.

Querying the number of **maxSession** and **sessionCount** in the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{
```

Generally on local machine the <LINK\_TO\_GRAPHQL\_ENDPOINT> would be  
http://localhost:4444/graphql

Querying all details for session, node and the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"g
```

Query for getting the current session count in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"g
```

Query for getting the max session count in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"g
```

Query for getting all session details for all nodes in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"s
```

Query to get slot information for all sessions in each Node in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"s
```

Query to get session information for a given session:

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"s
```

Querying the capabilities of each node in the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{
```

Querying the status of each node in the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{
```

Querying the URI of each node and the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{
```



Query for getting the current requests in the New Session Queue:

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"s
```

Query for getting the New Session Queue size :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":{"g
```

# 6.3 - Grid endpoints

## Grid

### Grid Status

Grid status provides the current state of the Grid. It consists of details about every registered Node. For every Node, the status includes information regarding Node availability, sessions, and slots.

```
cURL GET 'http://localhost:4444/status'
```

In the Standalone mode, the Grid URL is the Standalone server address.

In the Hub-Node mode, the Grid URL is the Hub server address.

In the fully distributed mode, the Grid URL is the Router server address.

Default URL for all the above modes is http://localhost:4444.

## Distributor

### Remove Node

To remove the Node from the Grid, use the cURL command enlisted below. It does not stop any ongoing session running on that Node. The Node continues running as it is unless explicitly killed. The Distributor is no longer aware of the Node and hence any matching new session request will not be forwarded to that Node.

In the Standalone mode, the Distributor URL is the Standalone server address.

In the Hub-Node mode, the Distributor URL is the Hub server address.

```
cURL --request DELETE 'http://localhost:4444/se/grid/distributor/node/
```

In the fully distributed mode, the URL is the Distributor server address.

```
cURL --request DELETE 'http://localhost:5553/se/grid/distributor/node/
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request DELETE 'http://<Distributor-URL>/se/grid/distributor/no
```

### Drain Node

Node drain command is for graceful node shutdown. Draining a Node stops the Node after all the ongoing sessions are complete. However, it does not accept any new session requests.

In the Standalone mode, the Distributor URL is the Standalone server address.

In the Hub-Node mode, the Distributor URL is the Hub server address.

```
cURL --request POST 'http://localhost:4444/se/grid/distributor/node/<r
```

In the fully distributed mode, the URL is the Distributor server address.

```
cURL --request POST 'http://localhost:5553/se/grid/distributor/node/<n
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request POST 'http://<Distributor-URL>/se/grid/distributor/node
```

# Node

The endpoints in this section are applicable for Hub-Node mode and fully distributed Grid mode where the Node runs independently. The default Node URL is http://localhost:5555 in case of one Node. In case of multiple Nodes, use [Grid status](#) to get all Node details and locate the Node address.

## Status

The Node status is essentially a health-check for the Node. Distributor pings the node status are regular intervals and updates the Grid Model accordingly. The status includes information regarding availability, sessions, and slots.

```
cURL --request GET 'http://localhost:5555/status'
```

## Drain

Distributor passes the [drain](#) command to the appropriate node identified by the node-id. To drain the Node directly, use the cuRL command enlisted below. Both endpoints are valid and produce the same result. Drain finishes the ongoing sessions before stopping the Node.

```
cURL --request POST 'http://localhost:5555/se/grid/node/drain' --heade
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request POST 'http://<node-URL>/se/grid/node/drain' --header 'X
```

## Check session owner

To check if a session belongs to a Node, use the cURL command enlisted below.

```
cURL --request GET 'http://localhost:5555/se/grid/node/owner/<session-
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request GET 'http://<node-URL>/se/grid/node/owner/<session-id>'
```

It will return true if the session belongs to the Node else it will return false.

## Delete session

Deleting the session terminates the WebDriver session, quits the driver and removes it from the active sessions map. Any request using the removed session-id or reusing the driver instance will throw an error.

```
cURL --request DELETE 'http://localhost:5555/se/grid/node/session/<ses
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request DELETE 'http://<node-URL>/se/grid/node/session/<session
```

# New Session Queue

## Clear New Session Queue

New Session Request Queue holds the new session requests. To clear the queue, use the cURL command enlisted below. Clearing the queue rejects all the requests in the queue. For each such request, the server returns an error response to the respective client. The result of the clear command is the total number of deleted requests.

In the Standalone mode, the Queue URL is the Standalone server address.

In the Hub-Node mode, the Queue URL is the Hub server address.

```
cURL --request DELETE 'http://localhost:4444/se/grid/newsessionqueue/c
```

In the fully distributed mode, the Queue URL is New Session Queue server address.

```
cURL --request DELETE 'http://localhost:5559/se/grid/newsessionqueue/c
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request DELETE 'http://<URL>/se/grid/newsessionqueue/queue' --h
```

## Get New Session Queue Requests

New Session Request Queue holds the new session requests. To get the current requests in the queue, use the cURL command enlisted below. The response returns the total number of requests in the queue and the request payloads.

In the Standalone mode, the Queue URL is the Standalone server address.

In the Hub-Node mode, the Queue URL is the Hub server address.

```
cURL --request GET 'http://localhost:4444/se/grid/newsessionqueue/queu
```

In the fully distributed mode, the Queue URL is New Session Queue server address.

```
cURL --request GET 'http://localhost:5559/se/grid/newsessionqueue/queu
```

# Selenium Level Sponsors



## Support the Selenium Project

Want to support the Selenium project? Learn more or view the full list of sponsors.

[LEARN MORE](#) ▶