Flutter Daemon

Overview

The `flutter` command-line tool supports a daemon server mode for use by IDEs and other tools.

```
flutter daemon
```

It runs a persistent, JSON-RPC based server to communicate with devices. IDEs and other tools can start the flutter tool in this mode and get device addition and removal notifications, as well as being able to programmatically start and stop apps on those devices.

A set of `flutter daemon` commands/events are also exposed via `flutter run --machine` and `flutter attach --machine` which allow IDEs and tools to launch and attach to Flutter applications and interact to send commands like Hot Reload. The command and events that are available in these modes are documented at the bottom of this document.

Protocol

The daemon speaks JSON-RPC to clients. It uses stdin and stdout as the transport protocol. To send a command to the server, create your command as a JSON-RPC message, encode it to JSON, surround the encoded text with square brackets, and write it as one line of text to the stdin of the process:

```
[{"method":"daemon.version","id":0}]
```

The response will come back as a single line from stdout:

```
[{"id":0,"result":"0.1.0"}]
```

All requests and responses should be wrapped in square brackets. This ensures that the communications are resilient to stray output in the stdout/stdin stream.

`id` is an opaque type to the server, but ids should be unique for the life of the server. A response to a particular command will contain the id that was passed in for that command.

Each command should have a `method` field. This is in the form '`domain.command`'.

Any params for that command should be passed in through a `params` field. Here's an example request/response for the `device.getDevices` method:

```
[{"method":"device.getDevices","id":2}]
```

```
[{"id":2,"result":[{"id":"702ABC1F-5EA5-4F83-84AB-6380CA91D39A","name":"iPhone
6","platform":"ios_x64","available":true}]}]
```

Events that come from the server will have an `event` field containing the type of event, along with a `params` field.

```
[{"event":"device.added","params":{"id":"1DD6786B-37D4-4355-AA15-B818A87A18B4","name":"iPhone XS
Max","platform":"ios","emulator":true,"ephemeral":false,"platformType":"ios","category":"mobile"}}]
```

Domains and Commands

daemon domain

daemon.version

The `version()` command responds with a String with the protocol version.

daemon.shutdown

The `shutdown()` command will terminate the flutter daemon. It is not necessary to call this before shutting down the daemon; it is perfectly acceptable to just kill the daemon process.

daemon.getSupportedPlatforms

The `getSupportedPlatforms()` command will enumerate all platforms supported by the project located at the provided `projectRoot`. It returns a Map with the key 'platforms' containing a List of strings which describe the set of all possibly supported platforms. Possible values include:

- android
- ios
- linux #experimental
- macos #experimental
- windows #experimental
- fuchsia #experimental
- web #experimental

Events

daemon.connected

The `daemon.connected` event is sent when the daemon starts. The `params` field will be a map with the following fields:

- `version`: The protocol version. This is the same version returned by the `version()` command.
- `pid`: The `pid` of the daemon process.

daemon.log

This is sent when user-facing output is received. The `params` field will be a map with the field `log`. The `log` field is a string with the output text. If the output indicates an error, an `error` boolean field will be present, and set to `true`.

daemon.showMessage

The `daemon.showMessage` event is sent by the daemon when some if would be useful to show a message to the user. This could be an error notification or a notification that some development tools are not configured or not installed. The JSON message will contain an `event` field with the value `daemon.showMessage`, and a `params` field containing a map with `level`, `title`, and `message` fields. The valid options for `level` are `info`, `warning`, and `error`.

It is up to the client to decide how best to display the message; for some clients, it may map well to a toast style notification. There is an implicit contract that the daemon will not send too many messages over a reasonable period of time.

daemon.logMessage

The `daemon.logMessage` event is sent whenever a log message is created - either a status level message or an error. The JSON message will contain an `event` field with the value `daemon.logMessage`, and a `params` field containing a map with `level`, `message`, and (optionally) `stackTrace` fields.

Generally, clients won't display content from `daemon.logMessage` events unless they're set to a more verbose output mode.

app domain

app.restart

The `restart()` restarts the given application. It returns a Map of `{ int code, String message }` to indicate success or failure in restarting the app. A `code` of `0` indicates success, and non-zero indicates a failure.

- `appId`: the id of a previously started app; this is required.
- `fullRestart`: optional; whether to do a full (rather than an incremental) restart of the application
- `reason`: optional; the reason for the full restart (eg. `save`, `manual`) for reporting purposes
- `pause`: optional; when doing a hot restart the isolate should enter a paused mode
- `debounce`: optional; whether to automatically debounce multiple requests sent in quick succession (this may introduce a short delay in processing the request)

app.reloadMethod (Removed)

This functionality was deprecated and removed after Flutter version 1.23.0

app.callServiceExtension

The `callServiceExtension()` allows clients to make arbitrary calls to service protocol extensions. It returns a `Map` - the result returned by the service protocol method.

- `appId` : the id of a previously started app; this is required.
- `methodName` : the name of the service protocol extension to invoke; this is required.
- `params` : an optional Map of parameters to pass to the service protocol extension.

app.detach

The `detach()` command takes one parameter, `appId` . It returns a `bool` to indicate success or failure in detaching from an app without stopping it.

- `appId` : the id of a previously started app; this is required.

app.stop

The `stop()` command takes one parameter, `appId` . It returns a `bool` to indicate success or failure in stopping an app.

- `appId` : the id of a previously started app; this is required.

Events

app.start

This is sent when an app is starting. The `params` field will be a map with the fields `appId` , `directory` , `deviceId` , and `launchMode` .

app.debugPort

This is sent when an observatory port is available for a started app. The `params` field will be a map with the fields `appId` , `port` , and `wsUri` . Clients should prefer using the `wsUri` field in preference to synthesizing a URI using the `port` field. An optional field, `baseUri` , is populated if a path prefix is required for setting breakpoints on the target device.

app.started

This is sent once the application launch process is complete and the app is either paused before main() (if `startPaused` is true) or main() has begun running. When attaching, this even will be fired once attached. The `params` field will be a map containing the field `appId` .

app.log

This is sent when output is logged for a running application. The `params` field will be a map with the fields `appId` and `log` . The `log` field is a string with the output text. If the output indicates an error, an `error` boolean field will be present, and set to `true` .

app.progress

This is sent when an operation starts and again when it stops. When an operation starts, the event contains the fields `id` , an opaque identifier, and `message` containing text describing the operation. When that same operation ends, the event contains the same `id` field value as when the operation started, along with a `finished` bool field with the value true, but no `message` field.

app.stop

This is sent when an app is stopped or detached from. The `params` field will be a map with the field `appId` .

app.webLaunchUrl

This is sent once a web application is being served and available for the user to access. The `params` field will be a map with a string `url` field and a boolean `launched` indicating whether the application has already been launched in a browser (this will generally be true for a browser device unless `--no-web-browser-launch` was used, and false for the headless `web-server` device).

Daemon-to-Editor Requests

These requests come *from* the Flutter daemon and should be responded to by the client/editor.

app.exposeUrl

This request is enabled only if `flutter run` is run with the `--web-allow-expose-url` flag.

This request is sent by the server when it has a local URL that needs to be exposed to the end-user. This is to support running on a remote machine where a URL (for example `http://localhost:1234`) may not be directly accessible to the end-user. With this URL clients can perform tunneling and then provide the tunneled URL back to Flutter so that it can be used in code that will be executed on the end-users machine (for example when a web application needs to be able to connect back to a service like the DWDS debugging service).

This request will only be sent if a web application was run in a mode that requires mapped URLs (such as using `--no-web-browser-launch` for browser devices or the headless `web-server` device when debugging).

The request will contain an `id` field and a `params` field that is a map containing a string `url` field.

The response should be sent using the same `id` as the request with a `result` map containing the mapped `url` (or the same URL in the case where the client does not need to perform any mapping).

device domain

device.getDevices

Return a list of all connected devices. The `params` field will be a List; each item is a map with the fields `id`, `name`, `platform`, `category`, `platformType`, `ephemeral`, `emulator` (a boolean) and `emulatorId`.

`category` is a string description of the kind of workflow the device supports. The current categories are "mobile", "web" and "desktop", or null if none.

`platformType` is a string description of the platform sub-folder the device supports. The current categories are "android", "ios", "linux", "macos", "fuchsia", "windows", and "web". These are kept in sync with the response from `daemon.getSupportedPlatforms`.

`ephemeral` is a boolean which indicates where the device needs to be manually connected to a development machine. For example, a physical Android device is ephemeral, but the "web" device (that is always present) is not.

`emulatorId` is a string ID that matches the ID from `getEmulators` to allow clients to match running devices to the emulators that started them (for example to hide emulators that are already running). This field is not guaranteed to be populated even if a device was spawned from an emulator as it may require a successful connection to the device to retrieve it. In the case of a failed connection or the device is not an emulator, this field will be null.

device.enable

Turn on device polling. This will poll for newly connected devices, and fire `device.added` and `device.removed` events.

device.disable

Turn off device polling.

device.forward

Forward a host port to a device port. This call takes two required arguments, `deviceId` and `devicePort`, and one optional argument, `hostPort`. If `hostPort` is not specified, the host port will be any available port.

This method returns a map with a `hostPort` fieldset.

device.unforward

Removed a forwarded port. It takes `deviceId`, `devicePort`, and `hostPort` as required arguments.

Events

device.added

This is sent when a device is connected (and polling has been enabled via `enable()`). The `params` field will be a map with the fields `id`, `name`, `platform`, `category`, `platformType`, `ephemeral`, and `emulator`. For more information on `platform`, `category`, `platformType`, and `ephemeral` see `device.getDevices`.

device.removed

This is sent when a device is disconnected (and polling has been enabled via `enable()`). The `params` field will be a map with the fields `id`, `name`, `platform`, `category`, `platformType`, `ephemeral`, and `emulator`. For more information on `platform`, `category`, `platformType`, and `ephemeral` see `device.getDevices`.

emulator domain

emulator.getEmulators

Return a list of all available emulators. The `params` field will be a List; each item is a map with the fields `id`, `name`, `category` and `platformType`. `category` and `platformType` values match the values described in `device.getDevices`.

emulator.launch

The `launch()` command allows launching an emulator/simulator by its `id`.

- `emulatorId`: the id of an emulator as returned by `getEmulators`.
- `coldBoot`: an optional boolean flag which indicates if the emulator should be cold booted. Only supported for android emulators, silently ignored if emulator type is not android.

emulator.create

The `create()` command creates a new Android emulator with an optional `name`.

- `name`: an optional name for this emulator

The returned `params` will contain:

- `success` - whether the emulator was successfully created
- `emulatorName` - the name of the emulator created; this will have been auto-generated if you did not supply one
- `error` - when `success = false`, a message explaining why the creation of the emulator failed

devtools domain

devtools.serve

The `serve()` command starts a DevTools server if one isn't already running. The return value will contain:

- `success` - whether the server started.
- `host` - the address host if the server successfully started.
- `port` - the port if the server successfully started.

'flutter run --machine' and 'flutter attach --machine'

When running `flutter run --machine` or `flutter attach --machine` the following subset of the daemon is available:

daemon domain

The following subset of the daemon domain is available in `flutter run --machine`. Refer to the documentation above for details.

- Commands

    - `version`
    - `shutdown`
- Events
    - `connected`
    - `log`
    - `logMessage`

app domain

The following subset of the app domain is available in `flutter run --machine`. Refer to the documentation above for details.

- Commands
    - `restart`
    - `callServiceExtension`
    - `detach`
    - `stop`
- Events
    - `start`
    - `debugPort`
    - `started`
    - `log`
    - `progress`
    - `stop`

Source

See the source for the daemon protocol and implementation.

Changelog

- 0.6.1: Added `coldBoot` option to `emulator.lauch` command.
- 0.6.0: Added `debounce` option to `app.restart` command.
- 0.5.3: Added `emulatorId` field to device.
- 0.5.2: Added `platformType` and `category` fields to emulator.
- 0.5.1: Added `platformType`, `ephemeral`, and `category` fields to device.
- 0.5.0: Added `daemon.getSupportedPlatforms` command
- 0.4.2: Added `app.detach` command
- 0.4.1: Added `flutter attach --machine`
- 0.4.0: Added `emulator.create` command
- 0.3.0: Added `daemon.connected` event at startup