hermes-inspector provides a bridge between the low-level debugging API exposed by Hermes and higher-level debugging protocols such as the Chrome DevTools protocol.

Targets

- chrome: classes that implement the Chrome DevTools Protocol adapter. Sits on top of classes provided by the inspector target.
- detail: utility classes and functions
- inspector: protocol-independent classes that sit on top of the low-level Hermes debugging API.

Testing

Tests are implemented using gtest. Debug logging is enabled for tests, and you can get debug logs to show even when tests are passing by running the test executable directly:

```
$ buck build //xplat/js/react-native-github/ReactCommon/hermes/inspector:chrome-tests
$ buck-out/gen/js/react-native-github/ReactCommon/hermes/inspector/chrome-tests
[...]
```

You can use standard gtest filters to only execute a particular set of tests:

```
$ buck-out/gen/js/react-native-github/ReactCommon/hermes/inspector/chrome-tests \
    --gtest_filter='ConnectionTests.testSetBreakpoint'
```

You can debug the tests using lldb or gdb:

```
$ lldb buck-out/gen/js/react-native-github/ReactCommon/hermes/inspector/chrome-tests
$ gdb buck-out/gen/js/react-native-github/ReactCommon/hermes/inspector/chrome-tests
```

Formatting

Make sure the code is formatted using the hermes clang-format rules before committing:

```
$ xplat/js/react-native-github/ReactCommon/hermes/inspector/tools/format
```

We follow the clang format rules used by the rest of the Hermes project.

Adding Support For New Message Types

To add support for a new Chrome DevTools protocol message, add the message you want to add to tools/message_types.txt, and re-run the message types generator:

```
$ xplat/js/react-native-github/ReactCommon/hermes/inspector/tools/run_msggen
```

This will generate C++ structs for the new message type in `chrome/MessageTypes.{h,cpp}`.

You'll then need to:

1. Implement a message handler for the new message type in `chrome::Connection`.
2. Implement a public API for the new message type in `Inspector`. This will most likely return a `folly::Future` that the message handler in (1) can use for chaining.
3. Implement a private API for the new message type in `Inspector` that performs the logic in Inspector's executor. (Inspector.cpp contains a comment explaining why the executor is necessary.)
4. Optionally, implement a method for the new message type in `InspectorState`. In most cases this is probably not necessary–one of the existing methods in `InspectorState` will work.

For a diff that illustrates these steps, take a look at D6601459.

Testing Integration With Nuclide and Apps

For now, the quickest way to use hermes-inspector in an app is with Eats. First, make sure the packager is running:

```
$ js1 run
```

Then, on Android, build the fbeats target:

```
$ buck install --run fbeats
```

On iOS, build the `//Apps/Internal/Eats:Eats` target:

```
$ buck install --run //Apps/Internal/Eats:Eats
```

You can also build `Eats` in Xcode using `arc focus` if you prefer an IDE:

```
$ arc focus --force-build \
    -b //Apps/Internal/Eats:Eats \
    cxxreact //xplat/hermes/API:HermesAPI //xplat/hermes/lib/VM:VM jsi \
    jsinspector hermes-inspector FBReactKit FBReactModule FBCatalystWrapper \
    //xplat/js:React //xplat/js/react-native-github:ReactInternal
```

For all the above commands, if you want to build the inspector `-O0` for better debug info, add the argument `--config hermes.build_mode=dbg`.

You should then be able to launch the app and see it listed in the list of Mobile JS contexts in the Nuclide debugger.