Meteor Tool

This part of the code-base is Meteor's CLI tool that reads commands from the user, builds the application, adds runtime code, provides an interface to Meteor Services (Accounts, Packages, Build Farms, deployments, etc).

The Meteor Tool is designed to be the "minimal kernel" as most of the functionality that goes into a typical Meteor app can pulled from core and 3rd-party packages.

Getting set up

Using Meteor in development is very simple. If Meteor spots that it is running from a Git checkout folder (having a `.git` directory), it will run in dev mode, download `npm` dependencies dynamically and pull the latest `dev_bundle` on the first run.

`dev_bundle` is a tarball with prebuilt binaries (`node` , `npm` , `mongod` , etc) and npm modules necessary for the Meteor tool. `dev_bundle` s are versioned and are built with a script in the `admin` directory. It is commonly built on Jenkins.

Usually it doesn't take long to get a new `dev_bundle` but if you are on a spotty network or switching between branches referencing different versions often, you can set the environment variable that will cache all downloaded versions indefinitely:

```
set SAVE_DEV_BUNDLE_TARBALL=t
```

You can also run `./meteor --get-ready` to install all npm dependencies for the tool.

Usually, the `meteor` script can download a new dev-bundle without any dependencies installed, but on Windows, it requires `7z` to be in the path for unpacking of a tarball. (Get 7-zip here)

Testing

Since the tool is a node app, it is not testable with general Meteor testing tools such as Tinytest. Instead the home-grown system "self test" is used.

"Self test" is a testing library that is focused on testing the CLI interactions and is rather an end-to-end testing tool (not a unit-testing tool). Albeit, it is often used for testing individual functions.

Besides monitoring the process output, "self test" is capable of mocking the package catalog and running from template apps.

The asserting syntax of "self test" is rather unusual since it operates on the process's stdout/stderr output after the process has run (not in real-time). A lot of assertions depend on timeouts and waits.

To run all tests, run the following:

```
# download all npm dependencies, etc
./meteor --get-ready

# set the multiplier for time-outs
set TIMEOUT_SCALE_FACTOR=3
```

```
# run the tests
./meteor self-test
```

Note, the scale factor for time-outs might be different depending on the hardware, but 3 is a safe choice for automation.

To quickly run an individual test or a group, pass a regular expression as an argument, it will be matched against test names:

```
./meteor self-test "login.*"
```

You can also run a particular file, or list all tests matching certain pattern, run with puppeteer (default), phantom or browserstack. See more at `./meteor help self-test`.

If you want to learn how to write a self-test, see the `tool-testing` subdirectory.

Profiling

Profiling is done in an ad-hoc way but it works well enough to spot obvious differences in things like build performance.

To enable profiling, set the environment variable to a "cut off" point, which is 100ms by default.

```
set METEOR_PROFILE=200
```

In this case, the reporter will only print calls that took more than 200ms to complete.

Internally, every profiled function should be wrapped into a `Profile(fn)` call. The entry point should be started explicitly with the `Profile.run()` call. Otherwise, it won't start measuring anything.

Debugging

Currently, to debug the tool with `node-inspector`, you can set the `TOOL_NODE_FLAGS` environment variable to be `--debug` or `--debug-brk`. This will modify the `meteor` bash script and run the tool with debugging enabled. The debugger will be listening to port 5858 by default, but it could be changed using the notation `--debug=6060` or `--debug-brk=6060`. Note that `node-inspector` should be compatible with the `node` version in the `dev_bundle`.

Next, start `node-inspector` from your checkout by going to `path/to/your/meteor/dev_bundle/lib/node_modules/node-inspector/bin` and run `inspector.js`. This will tell you the URL of the node inspector. If used with `-- debug-brk`, the script will pause on the first line.

In order to debug the test apps that `self-test` will spawn, the env variable `SELF_TEST_TOOL_NODE_FLAGS` could be used the same way `TOOL_NODE_FLAGS` is used. If you are setting the env variable `SELF_TEST_TOOL_NODE_FLAGS` with `TOOL_NODE_FLAGS`, consider specifying a custom port, as they could collide trying to listen to the same port. To set a custom port, you could set the variable in the following manner `SELF_TEST_TOOL_NODE_FLAGS="--debug-brk=5859"` and the debugger will listen to the port 5859 and not the default 5858.

Development

The entry-point of the tools code is in `index.js` .

Devel vs Prod environment

The Meteor Tool code has two modes of running:

- from local checkout for development
- from a production release installed by running `npm install -g meteor` .

There are two different `meteor` / `meteor.bat` starting scripts in development and production. The production one is written by the packaging code.

In addition to that, the checkout version stores its own catalog (`.meteor` dir) and the production version keeps it in `~/.meteor` .

When the release is published (with `./meteor publish-release --from-checkout`), the files committed into Git are copied in a compiled form into the built package (Isopack). You can find the list of copied sub-trees in `Isopack#_writeTool` .

Buildmessage

Throughout the code-base, there is an extensive use of `buildmessage` , which is a custom try/catch/finally system with recovery. See `/tools/utils/buildmessage.md` for more details.

More information

For more information about a particular part of Meteor Tool, see subdirectories' README.md files and the top-level intro comments in the bigger files.