Isobuild

Isobuild is the build system used by the Meteor Tool. See the high level description for more.

Terms

The terms Isobuild operates on often have two names: internal names and public concepts.

- `packageSource` - an abstract representation of a package/app source with metadata
- `isopack` - a compiled version of a package/app
- `unibuild` - a part of an isopack for a specific target (browser, server, tool, etc)
- `isopackCache` - an abstract representation of cached isopacks on disk
- `build plugin` - a part of an isopack that plugs into the build process
- `linked file` - a wrapped file by linker
- `project context` - an object that has lots of metadata, mostly about the packages, catalogs, it is responsible for catalogs refreshes, pulling the relevant packages, resolving dependencies, etc.
- `js-analyze` - some piece of software that implements a set of JS parser features

How an app is built

The app is built by Bundler, "bundling" is the most high-level process that happens to the application code once it is loaded. Here is a more detailed time-line:

1. Create the "Project Context" out of the app directory. Project Context makes sure we know what package versions we need to use and prepares them for us.
2. The Bundler is given the Project Context and now it is Bundler's job to create the Package Source for the app, compile app's parts as they were packages, and then put everything together.
3. The Builder is ran to write the output files to disk.
4. While all the building was done, a WatchSet of the whole app is collected. It's returned to the caller. Also, files `star.json` and `program.json` are written.

Compiler

Takes care of compiling an individual package and returning an Isopack.

Compiler has a dependency on Bundler, as it needs to bundle the build plugins used to compile the input package source.

Compiler runs various compiler batch plugins and “source handlers” (the old-style build plugins) on the source. If there is a need to build any build plugins to use them, it would recursively build those.

Bundler

Builds an individual app or a build plugin (that appears to be just an app that is run in the context of the build).

Bundler introduces additional terms:

- `JsImage` - is a representation of a built App or a build plugin.
- `ClientTarget` and `ServerTarget` are representations of two separate types of “programs” in a built App.

There are commonly two important entry-points for Bundler:

- `buildJsImage` - build a build plugin
- `bundle` - the main function to bundle the application from a Project Context

For the actual compilation, Bundler often calls into Compiler. Other tasks that Bundler performs include:

- initiate a Package Source for the app
- run linters on the app
- run the application files through Linker in a special mode that allows the use of global variables
- add a special file called “global-imports” that explicitly puts all used packages’ symbols into the global namespace
- write compiled files to the right location with multiple Builder’s
- write a `star.json` file with metadata for the “star” (the app) and a `program.json` file for every target

Builder

Manages the files written to the filesystem.

Since the rebuilds of an app is something that occurs over and over again (in the development cycle), it makes sense to reuse the information about the files that didn’t change. Builder tries not to spend too much time writing files that remained the same over and over again.

It is even OK to do when an app process serving these files is still running, on Unix, the process can retain files by their inodes (not by file paths) and then once the process release them, the FS will clean up unlinked files.

Linker

A Meteor-specific transform. Wraps every file into a closure, creates “package local variables” and sets up the “global imports” to look like `var Minimongo =`

`Package.minimongo.Minimongo;`.

The process of linking is Meteor's substitute to other module loading solutions like `r.js` or `require.js`, or ES2015-style modules. Eventually, Meteor wants to transition to ES2015 modules for everything.

The way linker works for individual modules (packages in Meteor's case), is several things. Roughly, it can be separated into two phases "prelink" and "link" (both combined define "fullLink").

Prelink is something that can be done with a package in Isolation:

- create a closure around each file
- concatenate all files
- add comments of the original line number, file-headers - the metadata to be read by humans when viewed in a browser that doesn't support source-maps
- run a JS parser and figure out all the global variables used, out of those, pick the ones that are assigned in the package and form "Package-level" global variables by adding a `var` for them.

In the second part "link", the linker caller already knows what versions of linked module's dependencies are and what exports they provide for the module:

- create import strings for globals that come from dependencies' exports: `var Minimongo = Package.minimongo.Minimongo;`, if the module references `Minimongo`.

Historically, Meteor used to ship "prelinked" files in packages and then "link" them in the bundle time. Starting with Meteor 1.2 and Batch Plugins API, Meteor distributes source files, so they are "fullLinked" in bunlde time together.

Batch Build Plugins

In Meteor 1.2, the new Build Plugin APIs have been introduced. You can read more about them here: wiki.

The Build Plugins APIs register compilers, minifiers and linters. All of them are applied on different stages of the build process. Compilers are used by Compiler and Isopack methods. Linters are ran per Unibuild in `compiler.lint` and for the App in Bundler. Minifiers are ran by Bundler on its last stage.

The previous generation of build plugins (File Handlers) used to run on package publish (i.e., built before published). In the world of Build Plugins, they are compiled as part of the bundle process.

WatchSet

WatchSets are collected throughout the build process. The idea is: the WatchSet represents all the files that participate in the build process and also this list is used to notice any changes for rebuilds.

Working on the Isobuild codebase, it is important to understand the common usage pattern: the common WatchSet is passed down to functions as an argument. The functions on the bottom, add files to the WatchSet, mutating the passed argument.

In the end, the final WatchSet that contains merged information about every used file, is returned to the caller.