Hot Module Replacement (HMR) is a method of updating javascript modules within a running application. This reduces the feedback cycle while developing, so you can view and test changes quicker. In Meteor's implementation, the app can be updated before the build has even finished.

{% pullquote warning %} `hot-module-replacement` package was introduced in Meteor 2.0 {% endpullquote %}

To enable HMR for an app, it should use the `hot-module-replacement` package. HMR is currently not supported for packages, but packages can depend on the `hot-module-replacement` package to ensure access to the hot API. When a change can not be accepted with HMR, Meteor uses hot code push to update the app, as is normally done when HMR is not used.

HMR currently supports the modern web architecture. It is always disabled in other architectures and in production.

How the app is updated

While rebuilding a supported architecture, Meteor checks which files were modified, and sends the modified files to the client. The client then uses this process:

1. It checks if the modified module accepts or declines updates. If the module does neither, Meteor looks at the modules that imported it to see if they accept or decline the update, and then the modules that import those, and so on. If all paths it followed leads to modules that accept the update, it uses HMR. Otherwise, it uses hot code push.
2. Many js modules do things that have a long term effect on the app. They might create Tracker autoruns, register event listeners, or have UI components that have been rendered. Modules can register dispose handlers to clean up the old version of a module so it no longer affects the app. At this point in the update process, those dispose handlers are called.
3. Meteor runs the new version of the module, the modules that accepted the update, and all of the modules between them. This ensures the module's new exports are used. Because of this, usually the only modules that accept updates are the ones that have no exports or that have another way of updating the exports used by parent modules. At this point, there are two versions of these modules running, but the old version is no longer in use if it was disposed properly.

For HMR to work properly, the correct modules have to accept or decline updates, and dispose handlers have to be written. Fortunately, most apps do not need to manually do either. Instead, you can use integrations that automatically detect modules that can accept updates, and how to clean up specific types of modules. For example, the React integration is enabled by default in Meteor apps, and is able to automatically update React components.

API

The hot-module-replacement package provides an API to control how HMR is used. The API is available at `module.hot`. Since the API isn't always available (for example, in production or in architectures not supported by HMR), the code should make sure `module.hot` is defined before using it:

```
if (module.hot) {
  module.hot.accept();
}
```

In a future Meteor version, using the if statement will allow minifiers to remove this block when minifying for production.

Packages that use the `module.hot` api should use the `hot-module-reload` package to ensure access to the API.

{% apibox "module.hot#accept" %}

HMR reruns files that import the modified modules so the files use the new exports. In addition to configuring which modules can be updated with HMR, `module.hot.accept()` also controls how many files are re-ran. Meteor re-runs the files that import the modified modules, the files that import those files, and so on, until it reaches the modules that accepted the update. Because of this, usually the only modules that accept updates are ones that have no exports, are have another way of updating the exports used by its parent modules.

{% apibox "module.hot#decline" %}

{% apibox "module.hot#dispose" %}

The call back is run when this instance of the module will no longer be used. The main use is making sure the instance of the module no longer affects the app. Here is an example where we stop a Tracker computation:

```
import { setLocale } from '/imports/utils/locale';

const computation = Tracker.autorun(() => {
  const user = Meteor.user();

  if (user && user.locale) {
    setLocale(user.locale);
  }
});

if (module.hot) {
  module.hot.dispose(() => {
    computation.stop();
  });
}
```

If it did not stop the computation, each time the module is reran for HMR, there would be an additional computation. This can lead to unexpected behavior, especially if we've modified the computation function.

The callback receives a data object that can be mutated to store information for the new instance of the module. This can be used to preserve class instances, state, or other data. For example, this module will preserve the value of the color variable:

```

let color = 'blue';

export function getColor() {
  return color;
}

export function changeColor(newColor) {
  color = newColor;
}

if (module.hot) {
  if (module.hot.data) {
    color = module.hot.data.color;
  }

  module.hot.dispose(data => {
    data.color = color;
  });
}
```

When the module first runs, `module.hot.data` is null, so it leaves `color` set to blue. Eventually the app calls `changeColor` and sets the color to `purple`. If the module is re-ran, the old instance of the module stores the color in `data.color`. The new instance retrieves it from `module.hot.data.color`, and registers a new dispose handler for the next time it is re-run.

{% apibox "module.hot#data" %}

{% apibox "module.hot#onRequire" %}

This is used by some HMR integrations to detect files that can be automatically updated with HMR, and handle cleaning up the old module instances and migrating state. For example, React Fast Refresh uses this to find the modules that only export React components, and have those modules accept updates.

```
if (module.hot) {
  module.hot.onRequire({
    // requiredModule is the same object available in the
    // required module as `module`, including access to `module.hot`
    // and `module.exports`
    //
    // parentId is a string with the path of the module that
    // imported requiredModule.
    before(requiredModule, parentId) {
      // Anything returned here is available to the
      // after callback as the data parameter.
      return {
        importedBy: parentId,
        previouslyEvaluated: !requiredModule.loaded
      }
    },
    after(requiredModule, data) {
      if (!data.previouslyEvaluated) {
        console.log(`Finished evaluating ${requiredModule.id}`);
        console.log(`It was imported by ${data.importedBy}`);
        console.log(`Its exports are ${requiredModule.exports}`);
      }

      // canAcceptUpdates would look at the exports, and maybe the imports
      // to check if this module can safely be updated with HMR.
      if (requiredModule.hot && canAcceptUpdates(requiredModule)) {
        requiredModule.hot.accept();
      }
    }
  });
}
```