Searching for packages

There are a few ways to search for Meteor packages published to Atmosphere:

1. Search on the Atmosphere website.
2. Use `meteor search` from the command line.
3. Use a community package search website like Fastosphere.

The main Atmosphere website provides additional curation features like trending packages, package stars, and flags, but some of the other options can be faster if you're trying to find a specific package. For example, you can use `meteor show ostrio:flow-router-extra` from the command line to see the description of that package and different available versions.

Package naming

You may notice that, with the exception of Meteor platform packages, all packages on Atmosphere have a name of the form `prefix:package-name`. The prefix is the Meteor Developer username of the organization or user that published the package. Meteor uses such a convention for package naming to make sure that it's clear who has published a certain package, and to avoid an ad-hoc namespacing convention. Meteor platform packages do not have any `prefix:`.

Installing Atmosphere Packages

To install an Atmosphere package, you run `meteor add`:

```
meteor add ostrio:flow-router-extra
```

This will add the newest version of the desired package that is compatible with the other packages in your app. If you want to specify a particular version, you can specify it by adding a suffix to the package name like: `meteor add ostrio:flow-router-extra@3.5.0`.

Regardless of how you add the package to your app, its actual version will be tracked in the file at `.meteor/versions`. This means that anybody collaborating with you on the same app is guaranteed to have the same package versions as you. If you want to update to a newer version of a package after installing it, use `meteor update`. You can run `meteor update` without any arguments to update all packages and Meteor itself to their latest versions, or pass a specific package to update just that one, for example `meteor update ostrio:flow-router-extra`.

If your app is running when you add a new package, Meteor will automatically download it and restart your app for you.

> *The actual files for a given version of an Atmosphere package are stored in your local `~/.meteor/packages` directory.*

To see all the Atmosphere packages installed run:

```
meteor list
```

To remove an unwanted Atmosphere package run:

```
meteor remove ostrio:flow-router-extra
```

You can get more details on all the package commands in the Meteor Command line documentation.

Using Atmosphere Packages

To use an Atmosphere Package in your app you can import it with the `meteor/` prefix:

```
import { Mongo } from "meteor/mongo";
```

Typically a package will export one or more symbols, which you'll need to reference with the destructuring syntax. You can find these exported symbols by either looking in that package's `package.js` file for `api.export` calls or by looking in that package's main JavaScript file for ES2015 `export` calls like `export const packageName = 'package-name';` .

Sometimes a package will have no exports and have side effects when included in your app. In such cases you don't need to import the package at all after installing.

> *For backwards compatibility with Meteor 1.2 and early releases, Meteor by default makes available directly to your app all symbols referenced in `api.export` in any packages you have installed. However, it is recommended that you import these symbols first before using them.*

Importing styles from Atmosphere packages

Using any of Meteor's supported CSS pre-processors you can import other style files using the `{package-name}` syntax as long as those files are designated to be lazily evaluated as "import" files. To get more details on how to determine this see CSS source versus import files.

```
@import '{prefix:package-name}/buttons/styles.import.less';
```

> *CSS files in an Atmosphere package are declared with `api.addFiles` , and therefore will be eagerly evaluated by default, and then bundled with all the other CSS in your app.*

Peer npm dependencies

Atmosphere packages can ship with contained npm dependencies, in which case you don't need to do anything to make them work. However, some Atmosphere packages will expect that you have installed certain "peer" npm dependencies in your application.

Typically the package will warn you if you have not done so. For example, if you install the `react-meteor-data` package into your app, you'll also need to install the `react` and the `react-addons-pure-render-mixin` packages:

```
meteor npm install --save react react-addons-pure-render-mixin
meteor add react-meteor-data
```

Atmosphere package namespacing

Each Atmosphere package that you use in your app exists in its own separate namespace, meaning that it sees only its own global variables and any variables provided by the packages that it specifically uses. When a top-level variable is defined in a package, it is either declared with local scope or package scope.

```
/**
 * local scope - this variable is not visible outside of the block it is
 * declared in and other packages and your app won't see it
 */
const alicePerson = {name: "alice"};

/**
 * package scope - this variable is visible to every file inside of the
 * package where it is declared and to your app
 */
bobPerson = {name: "bob"};
```

Notice that this is just the normal JavaScript syntax for declaring a variable that is local or global. Meteor scans your source code for global variable assignments and generates a wrapper that makes sure that your globals don't escape their appropriate namespace.

In addition to local scope and package scope, there are also package exports. A package export is a "pseudo global" variable that a package makes available for you to use when you install that package. For example, the `email` package exports the `Email` variable. If your app uses the `email` package (and *only* if it uses the `email` package!) then your app can access the `Email` symbol and you can call `Email.send`. Most packages have only one export, but some packages might have two or three (for example, a package that provides several classes that work together).

> *It is recommended that you use the `ecmascript` package and first call `import { Email } from 'meteor/email';` before calling `Email.send` in your app. It is also recommended that package developers now use ES2015 `export` from their main JavaScript file instead of `api.export`.*

Your app sees only the exports of the packages that you use directly. If you use package A, and package A uses package B, then you only see package A's exports. Package B's exports don't "leak" into your namespace just because you used package A. Each app or package only sees their own globals plus the APIs of the packages that they specifically use and depend upon.