OpenSphere Documentation

Release latest

William Wall, Kevin Schmidt

Jun 30, 2022

Developer Documentation

OpenSphere is a pluggable, single-page, GIS web application that supports both 2D and 3D views. It supports hooking up to many common servers and formats such as ArcGIS, Geoserver (and other OGC WMS/WFS services), XYZ, TMS, KML, GeoJSON, Shapefiles, CSVs, and more! Other features include animation of both raster and vector data, import and export of various formats, saving files and layers between sessions, and much more!

Check out the master branch here.

The code is open source, and available on GitHub.

CHAPTER 1

Contributing

1.1 Questions

Questions, design discussions, and roadmap talks should all be directed to our forum.

1.2 Bug Reports

Please use the GitHub Issue Tracker. As always, please search first to see if your issue has already been submitted. A minimum code sample to reproduce is also helpful.

1.3 Learning the Code

Our API contains many extensions to the OpenLayers API, so it would help to get familiar with Layers, Sources, Features, and Geometries from some of their examples and documentation.

1.4 Contributing Code

Our *Getting Started* guide has an overview of the build system, development, and testing the application in both compiled and debug modes.

We prefer that all contributions take place through pull requests. Please ensure that your pull request passes our pull request checklist.

CHAPTER 2

Getting Started

2.1 Setup

2.1.1 Prerequisites

- git
- Java 1.7.0+
- Node (Maintenance LTS) and npm
- Yarn
- Python
- Sphinx (optional, used to generate these docs)
- POSIX-compatible shell environment
  - Along with `cat`, `cp`, `echo`, `grep`, `perl`, `xargs`
- Chrome Browser (59+ required in default tests; plus you probably want this for development)
- Firefox Browser (57+ required in default tests)

Windows developers see Windows Development

Ensure that the executables `git`, `node`, and `java` are in your `PATH`.

2.1.2 Workspace Setup

Yarn

Clone opensphere-yarn-workspace, and change directory into `opensphere-yarn-workspace/workspace`.

Clone opensphere, then run `yarn install`.

NPM

(Not necessary if you followed the yarn instructions above)

Clone opensphere, then change directory to the clone. Run `npm install`.

Linking

If you are working on several plugins and config projects, you may end up with a workspace like:

```
workspace/
  opensphere/
  opensphere-config-developer/
  some-common-lib/
  opensphere-plugin-x/
  opensphere-plugin-y/
```

`npm link` is designed to help with this, but can get cumbersome to maintain manually with many projects. We recommend Yarn with opensphere-yarn-workspace to automate the links. If you prefer to use NPM, npm-workspace can be used instead.

Serving the application

While not required, we highly recommend setting up nginx or Apache httpd to serve up your workspace from your local machine. This will allow you to easily proxy remote servers and mock up services to develop against in addition to serving the application exactly as it will be served from production rather than accessing it via a file path in the browser or serving it with a node-based server.

You can also use the `start-server` target, for example:

`yarn start-server` or `npm run start-server`

2.2 The Build

Note: The build does not work natively in Windows. See Windows Development for instructions.

OpenSphere has all of its build targets as npm scripts. Therefore you can run any particular target by running:

```
$ npm run <target>
```

The most common targets are:

```
$ npm run build               # generates the production application
$ npm run dev                 # generates the dev application and runs webpack in␣
↪watch mode
$ npm run test                # runs the unit tests
$ npm run test:debug          # runs the unit tests with a configuration more suited␣
↪to debugging
$ npm run guide               # generates this documentation
$ npm run build:webpack-dev   # runs webpack in watch mode, for development
```

Each target runs its individual pieces through npm scripts as well. Several of those pieces are highly useful when run by themselves just to see if you fixed an error in that part of the build before restarting the entire thing.

```
$ npm run lint              # runs the linter to check code style
$ npm run compile:resolve   # runs the resolver to check dependency/plugin/config␣
↪resolution
$ npm run compile:gcc       # runs the google-closure-compiler to produce the compiled␣
↪JS
$ npm run compile:css       # runs sass to produce the minified/combined css
```

If you are using yarn (recommended), replace `npm run` with `yarn` in those targets.

2.2.1 The Resolver

opensphere-build-resolver runs through all of an application's dependencies, plugins (opensphere-plugin-x), or config projects (opensphere-config-y) and then the resolver's plugins produce arguments for the compiler, arguments for sass, page templates for conversion, and more! All of these files are written to the `.build` directory and used later in the build.

2.2.2 Webpack

OpenSphere's source is bundled using webpack and the closure-webpack-plugin. The plugin allows webpack to identify Google Closure files using `goog.declareModuleId`, `goog.module`, and `goog.provide` as build dependencies.

Webpack will also resolve ES modules and CommonJS modules imported with `require`. Modules should be imported using the Webpack/Node resolution method, with paths relative to the package containing the module.

Example:

```
const theModule = require('some-package/path/to/module');
```

Note: OpenSphere's webpack configuration can be found in `opensphere/webpack.config.js`.

2.2.3 The Google Closure Compiler

Use of the Closure Compiler has been limited among the open source community. However, unlike other projects which produce minified Javascript, the Closure Compiler is a true compiler. It does type checking, optimizations, and dead code removal. Type checking is essential to any large project, and the other optimizations allow our compiled code (in some cases) to perform three times better than our unminified code.

We use the compiler's `ADVANCED` compilation level, which is described in detail here. Also check out the annotations available for the compiler.

Because the Closure Compiler does so much more than just minification, the build takes a non-trivial amount of time to run. To help with developer productivity, we have produced a build system which does not need to be rerun when files change. Instead, it only needs to be run when files are added or dependencies change.

Some of the intricacies from using the compiler are documented in the *Compiler Caveats* section below.

2.2.4 The Development Build

To support various module types in a development build of the application, webpack bundles all source into a single file. This file includes source maps so individual source files can be viewed within the browser's developer tools.

The `index-template.html` and its corresponding `index.js` file define how the main page is packaged up by opensphere-build-index. That script produces `opensphere/index.html`, which is the root document for the dev build. It loads all of the vendor scripts and CSS in addition to the application bundle produced by webpack.

If you set up nginx or httpd as recommended above, accessing it might be accomplished by pointing your browser at http://localhost:8080/workspace/opensphere

Running `npm run dev` will generate the development application, and runs webpack in watch mode. Webpack will watch all dependencies for changes and rebuild the application when needed. While webpack is running, you can make changes to files in the workspace and pick them up on the page by refreshing it. The `npm run dev` script only has to be restarted if files are added or removed and cannot be resolved by webpack.

2.2.5 The Compiled Build

The compiled build output is available in `opensphere/dist/opensphere`. You will need to test your changes in both the development and compiled application, but generally compiled mode should be checked after you have largely completed the feature on which you are working. It does contain source maps for debugging, and also loads much quicker since all the code is compiled and minified to a smaller file.

2.3 Testing

All of our unit tests for opensphere are written in Jasmine and run with karma via `npm test`. Detailed coverage reports are available in `.build/test/coverage`. If you are writing a plugin or standalone application, you are free to use whatever testing framework you like, but you'll get more for free if you use what we've set up for you already. If you want to switch out Jasmine with something else (or a newer version of Jasmine), that should also be doable.

Any contributions to OpenSphere should avoid breaking current tests and should include new tests that fully cover the changed areas.

2.4 Git Commits

When making local commits, there are checks (implemented as git pre-commit hooks) to verify that your commit message matches the Conventional Commits conventions. Basically, you need use the form `<type>(<scope>): <subject>`, for example something like: `fix(docs): Updated Getting Started to describe git commits`. The valid types are: `feat`, `fix`, `docs`, `style`, `refactor`, `perf`, `test`, `build`, `ci`, `chore` and `revert`. Scope is optional, and should cover the particular part of opensphere that you are working on.

If your change is an API break, or would otherwise affect external projects, please add a `BREAKING CHANGE:` part to the commit message body (per conventions) that describes what external users need to do to adapt to the change.

2.5 Developing plugins

See our plugin guide to get started developing plugins.

2.6 Using OpenSphere as a library

See our application guide to get started using OpenSphere as a library for your own application.

2.7 Building the Read the Docs Guide

When modifying this guide, we recommend building it locally to ensure there are no errors/warnings in the build, and that everything displays correctly. The guide is built using Sphinx and the Read the Docs theme, which requires Python to install. To install the build dependencies:

```
pip install sphinx sphinx_rtd_theme sphinx-autobuild
```

Once dependencies are installed, generate the guide with `npm run guide`. The output will be available in `docs/_build/html`.

If you would like to automatically rebuild the guide as files change, use `npm run guide:auto`. This starts the `sphinx-autobuild` application to monitor the `docs` directory for changes and update the documentation accordingly. It also starts a live reload enabled web server to view changes as you make them, accessible at http://127.0.0.1:8000.

2.8 Compiler Caveats

The compiler will attempt to minify/rename any symbol it can. For the most part, this is preferred. However, when working with Angular templates, the variable/function names used in the HTML template will not be replaced and the HTML symbol will not match the JS symbol. To fix this, we use `@export` on symbols we do not want to rename.

Broken Example:

```
/**
 * A controller for an Angular directive.
 */
class MyController {
  /**
   * @param {!angular.Scope} $scope The Angular scope.
   * @ngInject
   */
  constructor($scope) {
    /**
     * This property will be renamed without @export.
     * @type {number}
     */
    this.value = 3;
  }

  /**
   * This function will be renamed without @export.
   * @param {number} value
   */
  isPositive(value) {
    return value > 0;
  }
}
```

```
<!-- Angular template -->
<span ng-show="ctrl.isPositive(ctrl.value)">{{ctrl.value}} is positive</span>
```

This will work great in development mode (no minification), but will fail in compiled mode. To fix this, we need to ensure that the compiled build does not minify the two items we used in the template.

Fixed Example:

```
/**
 * A controller for an Angular directive.
 */
class MyController {
  /**
   * @param {!angular.Scope} $scope The Angular scope.
   * @ngInject
   */
  constructor($scope) {
    /**
     * This property will not be renamed.
     * @type {number}
     * @export
     */
    this.value = 3;
  }

  /**
   * This function will not be renamed.
   * @param {number} value
   * @export
   */
  isPositive(value) {
    return value > 0;
  }
}
```

```
<!-- Angular template -->
<span ng-show="ctrl.isPositive(ctrl.value)">{{ctrl.value}} is positive</span>
```

Now it works in compiled mode! Note that UI templates is not the only place where `@export` is useful. It is useful wherever you want to have the compiler skip minification.

CHAPTER 3

Windows Development

Our build does not run natively on Windows due to its requirement of a POSIX shell and some of the core utilities such as `cat`, `cp`, `echo`, ect. However, you can build and develop OpenSphere on Windows with any of the following:

- Git Bash (included with Git for Windows)
- Cygwin
- Windows Subsystem for Linux
- Docker
- Linux VM

These instructions will take the simplest approach by using Git Bash.

3.1 Prerequisites

Install git, java, node, and yarn. We recommend using Chocolatey, a package manager for Windows. After installing Chocolatey, run the following in a command prompt as an administrator:

```
choco install git jre8 nvm yarn
refreshenv
nvm install 10.16.2
nvm use 10.16.2
```

Now we will check our work:

```
git --version
java -version
node --version
npm --version
yarn --version
```

Git for Windows installs "Git Bash". Search for it in the Start Menu and fire it up.

3.2 npm

Tell the NPM script runner to use BASH rather than whatever it typically uses on Windows.

```
npm config set script-shell "C:/Program Files/git/bin/bash.exe"
```

If you have other node projects on your machine and do not wish for this to pollute them, then consider adding that configuration to a `.npmrc` file local to the project.

3.3 git

Fix your line ending configuration for git (necessary if using Git Bash but not if you are using a full POSIX environment such as Cygwin):

```
git config core.autocrlf input
```

3.4 OpenSphere

Now set up the project workspace and clone the project:

```
cd <your workspace directory>
git clone https://github.com/ngageoint/opensphere-yarn-workspace
cd opensphere-yarn-workspace/workspace
git clone https://github.com/ngageoint/opensphere
yarn install
cd opensphere
```

Now build:

```
yarn build
# or
npm run build
```

Note: Yarn has a bug involving the use of the script-shell config with some older combinations of yarn/node. Please ensure that you are using the latest Yarn and also Node 12+.

CHAPTER 4

Application Guide

OpenSphere is typically extended to suit personal needs via Plugins, but a developer may want to completely replace the UI. To accomplish this, OpenSphere can be used as a library. This guide will outline the steps necessary to use OpenSphere as a library.

4.1 Application Package

Here we will walk through the necessary properties in the application's `package.json`. These properties are necessary for opensphere-build-resolver to determine how to build the application and package it for distribution.

4.1.1 Build

The `build` property is used to instruct the resolver during the build. For the example application, it looks like this:

Listing 1: `build` property for `package.json`

```
{
  "build": {
    "type": "app",
    "config": "config/settings.json",
    "index": "index.js",
    "scss": "scss/example.scss",
    "themes": [
      "overrides_default_compact",
      "overrides_slate_compact"
    ],
    "defineRoots": {
      "os.ROOT": "../opensphere/",
      "os.APP_ROOT": "./"
    },
    "gcc": {
      "define": [
```

(continues on next page)

(continued from previous page)

```
        "os.SETTINGS=config/settings.json",
        "os.config.appNs='example'"
      ],
      "entry_point": [
        "goog:ol.ext",
        "goog:exampleentry"
      ],
      "hide_warnings_for": [
        "/opensphere/"
      ]
    },
    "moduleDefines": {
      "my.module.PATH": "my-module/dist/my-module.min.js"
    }
  }
}
```

- `type`: This should be set to `app` so the resolver knows it's building an application.
- `config`: Where to find the application's main configuration file.
- `index`: The index file used by opensphere-build-index.
- `scss`: The root SCSS file for the application.
- `themes`: SCSS themes to include in the build. Available themes can be found in `opensphere/scss`.
- `defineRoots`: `goog.define` properties that need to be overridden from OpenSphere.
- `gcc`: Additional instructions for the Google Closure Compiler.
- `moduleDefines`: `goog.define` properties that should be resolved to a `node_module` path in uncompiled mode.

GCC

The `build.gcc` object has a number of properties available that affect what arguments are passed to the Google Closure Compiler.

- `define`: Overrides for `goog.define` calls within the application code. For the example app, we define where to find the settings file and replace the application namespace used for browser storage so it differs from OpenSphere.
- `entry_point`: The main `goog.declareModuleId` entry point for the application. This instructs the compiler on where to start resolving dependencies, and will include source files as needed by analyzing the dependency tree from there. The additional `ol.ext` entry point is a workaround to a Closure Compiler issue when using OpenLayers.
- `hide_warnings_for`: This hides errors/warnings from source files matching a pattern. This is useful if you don't care to see warnings from dependencies during the build.

Note: 2.x versions of the resolver will automatically add source files in the `<app>/src` directory. A future major release will likely change this behavior to explicitly define where to find source files with the `build.gcc.js` option. This ties in with the compiler `--js` flags that the resolver generates.

Module Defines

When using Yarn workspaces, dependencies may be hoisted to a parent `node_modules` directory. This makes the location of the module unpredictable and we must resolve it. This can be done directly for resources included via `index.js`, but any resources that need to be accessed programmatically will need their path resolved and defined for the uncompiled (debug) build.

This is accomplished by defining a `moduleDefines` property in the `build` section of the `package.json`. This property is a map of `goog.define` property names to the path of the resource being accessed. The path *must* begin with the module's package name (or it will not be resolved, resulting in an error), and any additional path is optional. In the example, we resolve the entire path to a minified JS file that could be lazily loaded in the application. The path could also be the root path of the package (just the package name), a directory, etc.

For the compiled build, set the default `goog.define` property value to the location you intend to copy the required resources to in `index.js`. For lazily loaded scripts, `index.js` should reference them in the `files` list instead of `scripts` so they aren't included in the `index.html`.

4.1.2 Directories

The `directories` property tells the resolver where certain resources for the application can be found. These are used by some of the resolver's plugins.

Listing 2: `build` property for `package.json`

```
{
  "directories": {
    "scss": "scss",
    "views": "views"
  }
}
```

For the example app, we're providing:

- `scss`: Tells the `scss` plugin where to find files to include in the `sass` build.
- `views`: Tells the `copy-views` plugin where to find HTML templates to copy for the distribution.

4.1.3 Scripts

These scripts are largely copied from OpenSphere's, and you will primarily use `yarn build` to build the application. The other scripts are all used by different parts of the build.

4.1.4 Dev Dependencies

The `devDependencies` section in the example app uses the same build tooling as OpenSphere, but you're welcome to change how your application is linted, tested, etc. The only dev dependencies required to build your code with OpenSphere's are `opensphere-build-resolver`, `@ngageoint/closure-webpack-plugin`, and `webpack`.

It also uses `eslint` for linting, and `sass` to compile SCSS, but again these are optional. For generating a development build (the *index.html* in the root project directory), opensphere-build-closure-helper is also required.

The remaining dev dependencies are for project management tasks like git hooks and npm scripts.

4.1.5 Dependencies

The primary dependency here is `opensphere` as our app dependency. Closure and Openlayers are inherited dependencies from OpenSphere, and left implicit to ensure the same version is used.

The example app sticks with Angular 1.8 to maintain compatibility with the numerous directives offered by OpenSphere, as well as ease of compilation with the Closure Compiler. Other frameworks (React, Vue.js, etc) may be used, but compatibility with the compiler may be limited.

Modernizr is included so the library can be built from a local config file. This library is used by OpenSphere to detect browser capabilities, and thus is required by any application using OpenSphere.

4.2 Application Index

This guide will walk through the application's `index.js`. This file is used by opensphere-build-index to generate the application's *index.html* (both debug and compiled versions), and by opensphere-build-resolver's `resources` plugin to identify/copy resources for the distribution. Everything in this file is documented in the README, but we'll quickly go over everything here.

4.2.1 Exports

The `module.exports` must provide the following:

- `appVersion`: Path to the application's version directory. This is output to `.build/version` by the build, so we read it from there.
- `packageVersion`: The version number for the application, typically read from the `package.json`.
- `basePath`: The base directory of the project, typically `__dirname`.
- `distPath`: The base directory of the distribution, typically `dist/<app>`. This is treated as relative to `basePath`.
- `templates`: HTML template files to process. This will generally be an `index-template.html` used to generate an `index.html`, but other templates can be specified as well.
- `debugCss`: Path to the compiled debug CSS output file.
- `debugJs`: Path to the debug JS output file.
- `compiledCss`: Path to the compiled CSS output file.
- `compiledJs`: Path to the compiled JS output file.

Debug JavaScript files will be determined by opensphere-build-closure-helper and automatically added to the `index.html` in the base directory.

4.2.2 Resources

Each template has a `resources` array that identifies what to include in the distribution. This should be an array of objects with the following properties:

- `source`: **[required]** Base path to locate specified resoures.
- `target`: **[required]** Base path under `appVersion` to copy specified resources.
- `scripts`: *(optional)* JavaScript files to copy *and* include as `<script>` tags in the HTML file.

- `css`: *(optional)* CSS files to copy *and* include as `<link>` tags in the HTML file.
- `files`: *(optional)* Additional files to copy, but not include. These are typically files referenced by the application.

Note: Resources often come from npm dependencies and are located in the `node_modules` folder. When using a Yarn workspace, this folder may exist at various levels in the folder structure. opensphere-build-resolver provides a utility function, `resolveModulePath`, that will locate these modules for you.

4.3 Application Structure

This guide explains the general structure of the OpenSphere example application, and how it wires up some basics needed to use the OpenSphere map.

4.3.1 Closure Entry Point

In the *Application Package*, we saw how the Closure Compiler entry point was set to `exampleentry`. This namespace is provided by exampleentry.js, which is responsible for initializing the application.

The first few lines in the initialization routine are fairly simple. We first configure the app's primary Angular module, example.Module, with a route provider that will load our `example-main` directive (the main app) when Angular is bootstrapped. Then we configure OpenSphere's request stack to use the default set of request handlers. You can customize which of these are loaded if you'd like, but the default set will handle:

- Local files in File Storage
- Same domain requests
- Cross-domain requests (supports CORS, with or without credentials)
- Proxy requests (requires a proxy to be configured in settings)

Next we initialize settings. OpenSphere stores application settings using the browsers IndexedDB API, with a fallback to localStorage if that isn't available. To ensure settings are loaded and available before the application tries to access them, we manually bootstrap Angular after they're loaded.

The example app creates a Settings Initializer that extends the base class from OpenSphere to override the root Angular `ng-app` module with example.Module. This initializer loads settings from storage, then bootstraps the module.

4.3.2 Index

The index template for our example application has two critical pieces that pair with the initialization routine. The `#ng-app` div will be the bootstrap target for the settings initializer, and the `ng-view` directive inside of it will be the target of our configured `$routeProvider` service. This will inject our application directive into the page when Angular is bootstrapped.

Note: You can customize the selector used to bootstrap Angular with the `ngAppSelector` property on the settings initializer. The example uses the default of `#ng-app`.

The template also has an `ng-init` directive that will configure the application version string and the path to the distribution version directory, which is used to locate templates and other resources.

Note: OpenSphere's build separates `index.html` from other application resources, which are placed in a version directory. This is intended to allow web servers to cache `index.html` for a short amount of time, while caching other resources for a much longer time. When releasing a new version of the application, the new version directory can be deployed alongside the old and as users' `index.html` cache expires they will pick up the new version.

4.3.3 Main Directive

The main directive for the example application sets up the core systems defined in OpenSphere. Since the app intends to use the `map` directive, it must configure:

- A data manager to manage the Openlayers data sources added to the map
- Any map interactions and UI controls
- Which plugins are loaded

Interactions and Controls

Openlayers provides an interaction API that OpenSphere uses and extends for the Cesium 3D globe. The example app adds some simple keyboard/mouse interactions for pan, zoom, etc. More complex interactions are available for features like data selection, drawing geometries, drawing measurement lines, and others. To see some of the other available interactions, see os.map.interaction in OpenSphere.

OpenSphere also uses the Openlayers control API to add UI components to the map. The example app adds some basics for map zoom/rotation, and also for toggling between a 2D map (Openlayers) and 3D globe (Cesium). To see more interactions used by OpenSphere, take a look at os.control.

Plugins

OpenSphere is designed to be extended via plugins. In our example app, we add a few plugins that add some map-related features:

- Loading WMS/WFS layers from OGC servers
- Adding XYZ map layers
- Configuring a set of base maps
- Enabling the Cesium 3D globe
- Loading supported file types (CSV, KML, GeoJSON, GPX, SHP)

OpenSphere has a number of other plugins available. To browse what's available, take a look at OpenSphere's addPlugins call or the plugin directory in the source. For more information on creating new plugins, see the *Plugin Guide*.

4.4 Example Project

Each guide references a sample application project, opensphere-app-example. To set up this project, clone it to a Yarn Workspace alongside OpenSphere, `yarn install`, and `yarn build` from the `opensphere-app-example` directory.

CHAPTER 5

ES6 Guide

Note: Please provide any feedback on this guide in #810. Thank you!

OpenSphere has migrated all of its code to use ES modules/classes/etc. This guide covers some of the differences from the legacy Closure-style source, and some specifics on how OpenSphere uses these language constructs.

5.1 Using ES Modules in OpenSphere

Modules have a number of key differences from files using `goog.provide` that are important for developers to keep in mind.

5.1.1 Module Scope

Every module has its own local scope. This prevents polluting the global `window` context because all variables defined in the module are local by default.

Consider the following code at the root level of a JavaScript file:

```
const myString = 'Hello, World!';
```

If this were loaded in a normal script, `window.myString` would be set to the string `Hello, World!`. When loaded in a module, the variable is only accessible locally in the module. `window.myString` will still be `undefined`.

5.1.2 Module Exports

Note: This documentation is a basic export example. For more complete documentation, see the MDN import and export documentation.

To provide functions, classes, etc to external files, a module may use the `export` keyword to define what to expose from its local scope. To make the above string available externally, a module could do the following:

```
goog.declareModuleId('my.module');

export const myString = 'Hello, World!';

exports = {myString};
```

Then to reference that string in another module:

```
import {myString} from './path/to/my/module.js';

console.log(myString);
```

Requiring an ES Module in Legacy Code

In a legacy `goog.provide` file, `goog.require` always returns `null` and should never be assigned to a variable. Doing so would pollute the global context by adding the variable name to `window`. To reference an ES module from a `goog.provide` type file, you must use `goog.module.get` in a restricted scope and assign the exports.

For example:

```
goog.provide('my.legacy');

goog.require('my.module');

my.legacy.printTheString = function() {
  const {myString} = goog.module.get('my.module');
  console.log(myString);
};
```

In a `goog.module` file, `goog.require` returns the exports so this can be simplified a bit.

```
goog.module('my.legacy');

const {myString} = goog.require('my.module');

const printTheString = function() {
  console.log(myString);
};

exports = {printTheString};
```

5.1.3 Type Only Imports

If an `import` or `goog.require` is only needed to access types in a module, use `goog.requireType`. This will only be used by the compiler for type checking and does not create a hard dependency on the required module. These calls will also be discarded from the compiled output.

```
// SomeEvent is a dependency and programmatically used in the file.
import SomeEvent from './path/to/someevent.js';

// The SomeEvent type is referenced in JSDoc, and is not a dependency.
const {default: SomeEvent} = goog.requireType('os.SomeEvent');
```

Note: When using `goog.requireType` with an ES module, Closure will assign the default export to a `default` property on the exports, and any named exports to like-named properties. This is why the above example reassigns the `default` property to a more friendly name. This is also necessary when using `goog.module.get` with an ES module.

5.1.4 Typedefs

`@typedef` declarations are only used by the compiler, but must be exported if they're used outside the file that declares them. Alternatively they can be moved to an extern to avoid the need for `goog.requireType` to use them.

```
/**
 * @typedef {{
 *   prop1: string,
 *   prop2: number
 * }}
 */
export let MyType;
```

5.2 Migrating from goog.modules to ES modules

The primary difference between a `goog.module` and an ES module is how imports/exports are defined. With `goog.module`, we solely use `goog.require` to import dependencies and use the `exports` keyword to assign either a single default export or an object listing named exports. ES modules use the `import` and `export` keywords, which are part of the JavaScript specification. MDN's JavaScript guide has an excellent reference on JavaScript modules with details on how these keywords work.

5.2.1 Transform Script

To convert `goog.module` files to ES modules, we'll use the `moduletoes6.js` transform in opensphere-jscodeshift. This transform is based on the Closure Compiler's Migrating from goog.modules to ES modules guide.

To run the transform, clone `opensphere-jscodeshift` to your workspace and `yarn upgrade`. Then run the transform on a `goog.module` file:

```
$ cd /path/to/opensphere-jscodeshift
$ yarn run shift -t src/transforms/es6/moduletoes6.js <target file or directory>
```

The transform can be run against a single file or a directory. In the case of a directory, it will recursively run against any `.js` files it finds under the base path. The transform will make the following changes:

- Replace the `goog.module` expression with `goog.declareModuleId`.
- Replace `exports` with inline `export` and `export default` declarations.

- If `export default` is used, creates a `<filename>.shim.js` file to maintain backwards compatibility with `const TheDefaultExport = goog.require(<module name>)`.
- Remove the `goog.module.declareLegacyNamespace` statement, if present.

5.2.2 goog.declareModuleId

To make ES modules accessible to `goog.module` or `goog.provide` files, the Closure Library provides the `goog.declareModuleId` function to declare a Closure module ID. This is analagous to `goog.module`, and makes the ES module's exports available via `goog.require` or `goog.module.get`.

Note: While `goog.declareModuleId` is not necessary for source files, it does allow tests to load module exports via `goog.module.get`. Unless tests are migrated to fully support loading modules, this statement should be included in every file.

5.2.3 Referencing ES Modules

The code used to reference an ES module varies based on the file type referencing the module. For these examples, assume we have the following ES modules in OpenSphere.

Listing 1: An index module with named exports at `src/os/index.js`.

```
goog.declareModuleId('os');

/**
 * A named export.
 * @type {number}
 */
export const MY_CONSTANT = 42;
```

Listing 2: A class with a default export at `src/os/myclass.js`.

```
goog.declareModuleId('os.MyClass');

/**
 * A default export.
 */
export default class MyClass {}
```

From an ES Module

To import these files from another ES module, use `import` statements with a path to the file. Within OpenSphere, use a relative path. From external projects, use a Node path. The file's `.js` extension is not required in the path, and for `index.js` files the path can end at the containing directory.

```
// From OpenSphere
import {MY_CONSTANT} from '../path/to/os';
import MyClass from '../path/to/os/myclass';

// From another project
```

(continues on next page)

(continued from previous page)

```
import {MY_CONSTANT} from 'opensphere/src/os';
import MyClass from 'opensphere/src/os/myclass';
```

From a Goog Module

To import these files from a `goog.module`, use `goog.require` assignments.

```
const {MY_CONSTANT} = goog.require('os');
const {default: MyClass} = goog.require('os.MyClass');
```

Note: The default export from an ES module will be assigned to the `default` property on the object returned by `goog.require`. This is why the transform script creates a shim file, so existing references to the module do not need to be updated. You do not need to create this shim for new files, simply use the above syntax to reference the default export properly.

From Legacy Files/Tests

To import the modules from a legacy file using `goog.provide` or from tests, use a combination of `goog.require` to add the dependency and `goog.module.get` to reference the module.

```
// Add a dependency on the module
goog.require('os');
goog.require('os.MyClass');

// Load the module
const {MY_CONSTANT} = goog.module.get('os');
const {default: MyClass} = goog.module.get('os.MyClass');
```

5.2.4 Legacy Namespaces

With `goog.module` files, the `goog.module.declareLegacyNamespace` function is called to export the module's namespace to the global `window` object. This function cannot be used in an ES module because it violates a core principle of modules, that they do not pollute the global scope. The transform script will remove this function from converted files, so prior to running the transform please ensure the module is no longer referenced using the global namespace.

5.3 Using ES6 Classes in OpenSphere

As part of the transition, OpenSphere is converting all Google Closure style classes to use the ES6 class syntax. An ES6 `class` is fundamentally the same as a class in Closure, which means the two can be intermingled to a degree.

5.3.1 External Documentation

The following are a few helpful guides/references for ES6 classes. It's recommended to start here if you do not already have an understanding of JavaScript classes.

- Introduction to ES6 classes

- MDN reference
- Simple ES6 guide

5.3.2 ES6 vs Closure

While ES6 classes and Closure classes are nearly the same under the hood, they are very different in syntax. This section outlines the key differences.

Closure Constructor

- Class name assigned to a constructor function.
- Requires the `@constructor` JSDoc annotation to inform the compiler that it's a class constructor.
- If it extends another class, the `@extends` annotation is also required.
- Parent constructor is invoked with `<class>.base(this, 'constructor', <args>)`.
- Class properties are added with `this.<property>`.
- Prototypal inheritance is established by calling `goog.inherits(<child>, <parent>)`.

```
/**
 * Description of the class.
 * @param {string} arg1 A string arg.
 * @param {number} arg2 A number arg.
 * @extends {os.MyParentClass}
 * @constructor
 */
os.MyClass = function(arg1, arg2) {
  os.MyClass.base(this, 'constructor', arg1);

  /**
   * Description of arg2.
   * @type {number}
   * @private
   */
  this.arg2_ = arg2;
};
goog.inherits(os.MyClass, os.MyParentClass);
```

ES6 Constructor

- Defined using the `class` keyword.
- Constructor is defined with the special `constructor` function. The `@constructor` annotation is not required.
- If it extends another class, the `extends` keyword is used. This establishes prototypal inheritance, and cues the compiler so `@extends` is not required. `@extends` is still used when providing generic types for a template (ie, `@extends {MyParentClass<MyType>}`).
- Parent constructor is invoked with `super(<args>)`.
- Class properties are added with `this.<property>`.

```
/**
 * Description of the class.
 */
class MyClass extends MyParentClass {
  /**
   * @param {string} arg1 A string arg.
   * @param {number} arg2 A number arg.
   */
  constructor(arg1, arg2) {
    super(arg1);

    /**
     * Description of arg2.
     * @type {number}
     * @private
     */
    this.arg2_ = arg2;
  }

  /**
   * @inheritDoc
   */
  setArg1(arg1) {
    super.setArg1(arg1);
    console.log('Set arg1 in parent.');
  }
}
```

Warning: An ES6 class can extend a Closure class, but not the reverse. Closure classes add properties under the hood for `goog.base` and ES6 classes will not have these. This means leaf classes must be refactored before their parents.

Member Functions

With Closure classes, member functions are added to the prototype. They reference the parent function using `<class>.base(this, '<function>', <args>)`.

```
/**
 * @inheritDoc
 */
os.MyClass.prototype.setArg1 = function(arg1) {
  os.MyClass.base(this, 'setArg1', arg1);
  console.log('Set arg1 in parent.');
};
```

In ES6, member functions are defined within the `class`. They reference the parent function using `super.<function>(<args>)`.

```
  /**
   * @inheritDoc
   */
  setArg1(arg1) {
    super.setArg1(arg1);
```

(continues on next page)

(continued from previous page)

```
    console.log('Set arg1 in parent.');
  }
```

Note: OpenSphere has historically used the `<get/set>PropName` pattern to get/set properties on a class. The get and set syntax has been around for awhile and works well with ES6 classes, but switching to it would be a breaking change.

Properties

Both class styles define their properties in the constructor in much the same manner.

The key difference for ES6 classes is that they are @struct by default, which prevents using bracket notation or adding properties outside the constructor. If either of those are needed (ie, using bracket notation to avoid property renaming), the class must have @unrestricted in the JSDoc.

```
/**
 * @unrestricted
 */
class MyClass {
  constructor() {
    // bracket notation to avoid compilation
    this['id'] = 1234;
  }
}
```

Singletons

Class instance singletons have historically been created using `goog.addSingletonGetter` which adds a `getInstance` function to the class. With ES6 classes and the local scope provided by modules, this is easy to do natively by adding a `static getInstance()` call to the class.

```
// store the instance in a module-scoped variable that can be externally referenced
// with MyClass.getInstance()
let instance;

class MyClass {
  constructor() {}

  static getInstance() {
    // do not create the instance until the first time this function is called
    if (!instance) {
      instance = new MyClass();
    }

    return instance;
  }
}
```

Constants

Constants on a class can be represented using a combination of the static and get keywords. This is a convenient way to define the constant on the class without needing to export the constant.

```
class MyClass {
  constructor() {}

  static get MY_CONSTANT() {
    return theConstant;
  }
}

// constant can be externally referenced with MyClass.MY_CONSTANT
const theConstant = 42;
```

5.4 Angular Directives in Modules

A primary use case of having multiple `goog.provide` statements per file is with Angular directives and their controller. OpenSphere prefers to pair the directive and controller within the same file given they are coupled to create the UI. This poses a problem for a backward-compatible transition to modules, given only one module name is allowed per file.

Our options were to split the original names into separate files, or change our approach to exposing the UI. We decided to use a shim to maintain compatibility with existing code. This seems to result in the best end product, and will be detailed in this guide.

The following example shows how the `map` directive was transitioned to a module.

Note: For a full example, see the map directive source here.

5.4.1 Creating the UI Module

- The `goog.module` value can use the original controller name, minus `Ctrl`. The name can be adjusted if needed, for example if this convention results in a name conflict with another class. If unsure about naming conflicts, we recommend replacing `Ctrl` with `UI` in the module name.
- Define and export the controller class as `Controller`, and the directive function as `directive`. This will ensure consistency across all UI's.

Listing 3: `src/os/ui/map.js`

```
goog.module('os.ui.MapUI');
goog.module.declareLegacyNamespace();

/**
 * Controller for the map directive.
 */
class Controller {
  // Controller class body
}
```

(continues on next page)

(continued from previous page)

```
/**
 * The map directive.
 * @return {angular.Directive}
 */
const directive = () => ({
  // Directive definition
});

// Export on the module as Controller and directive.
exports = {Controller, directive};
```

5.4.2 Using the Module

To reference the UI in `goog.module` files:

```
const MapUI = goog.require('os.ui.MapUI');
// reference the controller class as MapUI.Controller
// reference the directive function as MapUI.directive
```

Note: This intentionally uses the name convention `<class>UI` both for clarity that it's a UI where referenced, and the avoid shadowing the native `Map` object when assigned to a variable of the same name.

To reference the UI in legacy `goog.provide` files:

```
goog.require('os.ui.MapUI');
// reference the controller class as os.ui.MapUI.Controller
// reference the directive function as os.ui.MapUI.directive
```

5.4.3 Backward Compatibility Shim

When converting existing UI's to modules, we would like to avoid breaking changes where possible. To avoid a breaking change with a converted module, we'll create a shim to provide the old namespaces. The shim needs to accomplish a couple things:

- Make the old namespaces available to `goog.require` statements.
- Deprecate the old namespaces so developers are aware of the change.

Listing 4: `src/os/ui/map_shim.js`

```
goog.provide('os.ui.MapCtrl');
goog.provide('os.ui.mapDirective');

goog.require('os.ui.Map');

/**
 * @deprecated Please use goog.require('os.ui.Map').Controller instead.
 */
os.ui.MapCtrl = os.ui.Map.Controller;

/**
 * @deprecated Please use goog.require('os.ui.Map').directive instead.
```

(continues on next page)

(continued from previous page)

```
 */
os.ui.mapDirective = os.ui.Map.directive;
```

The shim will be maintained until all code referencing the old namespaces has been transitioned to directly reference the new module. After a suitable (TBD) amount of time has passed for developers to update their code, the shim will be deleted.

Note: New UI's do not need a shim because all references to them can be guaranteed to use the new format.

Warning: The deprecation warning will not appear as a result of `goog.require`. The Closure compiler will only issue a warning if the old directive/controller references are invoked in code.

5.5 Testing Goog Modules

When referencing a Closure or ES module in a test, the `goog.module.get` call must be used to retrieve the exports. This call should be made within a closure such as the `describe` function. The test will also need to use a bare `goog.require` on the module ID from `goog.declareModuleId` or `goog.module`.

```
// this test file is not a module, so
// const SomeModule = goog.require('package.SomeModule');
// will not work
goog.require('package.SomeModule');

describe('package.SomeModule', () => {
  const {default: SomeModule} = goog.module.get('package.SomeModule');

  it('should do stuff...', () => {
  });
});
```

5.6 External Documentation

Closure has several Wiki pages on `goog.module` and ES6/Closure that are all recommended reading to help understand this process.

- goog.module: an ES6 module like alternative to goog.provide
- Migrating from goog.module to ES6 modules
- Closure/ES6 interop
- ES6 and the Closure Library

CHAPTER 6

Plugin Guide

OpenSphere plugins can add new layer types, new file formats for importing and exporting, new provider or server types for listing available data from various services, new search implementations, new UI elements or windows, and much, much more!

6.1 File Type Plugin Guide

Here we will walk through creating a plugin for a new vector file type, georss-simple. There are many other examples of file type support within OpenSphere itself, so have a look here for more examples.

If you wish for this to be an external, separately released plugin, then fork opensphere-plugin-example and follow the instructions in its readme as a starting point.

If you wish for this to be a core plugin included with OpenSphere then simply begin adding your plugin code to `src/plugin/yourplugin`. However, if you are serious about getting your plugin included to the core project, please create an issue for it so we can discuss it.

6.1.1 Plugin

Add a basic plugin class.

Listing 1: `src/plugin/georss/georssplugin.js`

```
goog.declareModuleId('plugin.georss.GeoRSSPlugin');

import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import {ID} from './georss.js';

/**
 * Provides support for the GeoRSS format.
```

(continues on next page)

(continued from previous page)

```
 */
export default class GeoRSSPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    // plugin does not do anything yet
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new GeoRSSPlugin());
```

Internal Plugins If creating an internal plugin, ensure that `mainctrl.js` imports your plugin, or optionally, the external plugin method will also work.

External Plugins If creating an external plugin, ensure that `package.json` `build.gcc.entry_point` has `goog:plugin.georss.GeoRSSPlugin` in its list.

Run `yarn build` in OpenSphere (not in your plugin if it is external). It should build just fine but it does not do anything yet.

Just for good measure, let's test it.

Listing 2: `test/plugin/georss/georssplugin.test.js`

```
// os.mock sets up a bunch of basic opensphere APIs, like settings, which is
// used in our example plugin
goog.require('os.mock');
goog.require('plugin.georss.GeoRSSPlugin');

describe('plugin.georss.GeoRSSPlugin', function() {
  const {default: GeoRSSPlugin} = goog.module.get('plugin.georss.GeoRSSPlugin');

  it('should have the proper ID', function() {
    expect(new GeoRSSPlugin().id).toBe('georss');
  });

  it('should not throw an error', function() {
    var fn = function() {
      var p = new GeoRSSPlugin();
      p.init();
    };

    expect(fn).not.toThrow();
  });
});
```

Run `yarn test` in your plugin project if it is external (not in OpenSphere) to run its tests. For internal plugins, no directory change is necessary.

6.1.2 Parser

The first thing we need is a parser that can take the file and turn it into usable `ol.Feature` instances.

Listing 3: `src/plugin/georss/georssparser.js`

```
goog.declareModuleId('plugin.georss.GeoRSSParser');

import {PROJECTION} from 'opensphere/src/os/map/map.js';

const Feature = goog.require('ol.Feature');
const LineString = goog.require('ol.geom.LineString');
const Point = goog.require('ol.geom.Point');
const Polygon = goog.require('ol.geom.Polygon');
const {isDocument, parse} = goog.require('ol.xml');

const {default: IParser} = goog.requireType('os.parse.IParser');


/**
 * Parser for GeoRSS feeds
 * @implements {IParser<Feature>}
 * @template T
 * @constructor
 */
export default class GeoRSSParser {
  /**
   * Constructor.
   */
  constructor() {
    /**
     * @type {?Document}
     * @protected
     */
    this.document = null;

    /**
     * @type {?NodeList}
     * @protected
     */
    this.entries = null;

    /**
     * @type {number}
     * @protected
     */
    this.nextIndex = 0;
  }

  /**
   * @inheritDoc
   */
  setSource(source) {
    if (isDocument(source)) {
```

(continues on next page)

(continued from previous page)

```
      this.document = /** @type {Document} */ (source);
    } else if (typeof source === 'string') {
      this.document = parse(source);
    }

    if (this.document) {
      this.entries = this.document.querySelectorAll('entry');
    }
  }

  /**
   * @inheritDoc
   */
  cleanup() {
    this.document = null;
    this.entries = null;
    this.nextIndex = 0;
  }

  /**
   * @inheritDoc
   */
  hasNext() {
    return this.entries != null && this.entries.length > this.nextIndex;
  }

  /**
   * @inheritDoc
   */
  parseNext() {
    var nextEntry = this.entries[this.nextIndex++];
    var children = nextEntry.childNodes;
    var properties = {};

    for (var i = 0, n = children.length; i < n; i++) {
      var el = /** @type {Element} */ (children[i]);

      if (el.localName === 'link') {
        properties[el.localName] = el.getAttribute('href');
      } else if (el.namespaceURI === 'http://www.georss.org/georss') {
        var geom = parseGeometry(el);
        if (geom) {
          properties['geometry'] = geom;
        }
      } else {
        properties[el.localName] = el.textContent;
      }
    }

    return new Feature(properties);
  }
}

/**
 * @param {Element} el The element to parse
 * @return {ol.geom.Geometry|undefined} the geometry
 */
```

(continues on next page)

(continued from previous page)

```
export const parseGeometry = function(el) {
  switch (el.localName) {
    case 'point':
      return parsePoint(el);
    case 'line':
      return parseLine(el);
    case 'polygon':
      return parsePolygon(el);
    default:
      break;
  }
};

/**
 * @param {Element} el The element to parse
 * @return {Point|undefined} The point geometry
 */
const parsePoint = function(el) {
  var coords = parseCoords(el);

  if (!coords || coords.length === 0) {
    // no coords found!
    return;
  }

  return new Point(coords[0]);
};

/**
 * @param {Element} el The element to parse
 * @return {LineString|undefined} The line geometry
 * @private
 */
const parseLine = function(el) {
  var coords = parseCoords(el);

  if (!coords) {
    // no coords found!
    return;
  }

  if (coords.length < 2) {
    // need at least 2 coords for line!
    return;
  }

  return new LineString(coords);
};

/**
 * @param {Element} el The element to parse
 * @return {Polygon|undefined} The polygon geometry
 */
const parsePolygon = function(el) {
  var coords = parseCoords(el);

  if (!coords) {
```

(continues on next page)

(continued from previous page)

```
    // no coords found!
    return;
  }

  if (coords.length < 3) {
    // need at least 3 coords for polygon!
    return;
  }

  return new Polygon([coords]);
};

/**
 * @param {Element} el The element to parse
 * @return {Array<ol.Coordinate>|undefined} The array of coordinates
 */
const parseCoords = function(el) {
  var parts = el.textContent.trim().split(/\s+/);

  if (parts.length % 2 !== 0) {
    // odd amount of numbers, cannot produce pairs!
    return;
  }

  var coords = [];
  for (var i = 1, n = parts.length; i < n; i += 2) {
    var lat = parseFloat(parts[i - 1]);
    var lon = parseFloat(parts[i]);

    if (isNaN(lat) || isNaN(lon)) {
      // could not parse all lat/lons of coordinates!
      return;
    }

    var coord = [lon, lat];

    // convert to the application projection
    coords.push(ol.proj.fromLonLat(coord, PROJECTION));
  }

  return coords;
};
```

Whew. That was a lot for one step. It is not exhaustive, and a full implementation would want to support RSS in addition to Atom as well as the `<georss:elev>` tag. However, it still would not be complete without some tests.

Listing 4: `test/plugin/georss/georssparser.test.js`

```
goog.require('ol.xml');
goog.require('plugin.georss.GeoRSSParser');

describe('plugin.georss.GeoRSSParser', function() {
  const {parse} = goog.module.get('ol.xml');
  const {default: GeoRSSParser} = goog.module.get('plugin.georss.GeoRSSParser');

  it('should parse points correctly', function() {
```

(continues on next page)

(continued from previous page)

```
    var el = parse('<point>  40    -105  ' +
      '</point>').firstElementChild;

    var geom = GeoRSSParser.parseGeometry(el);

    expect(geom instanceof ol.geom.Point).toBe(true);
    expect(geom.getCoordinates()[0]).toBe(-105);
    expect(geom.getCoordinates()[1]).toBe(40);
  });

  it('should choose the first point if there is more than one', function() {
    var el = parse('<point>40 -105 50 -95</point>').firstElementChild;

    var geom = GeoRSSParser.parseGeometry(el);

    expect(geom instanceof ol.geom.Point).toBe(true);
    expect(geom.getCoordinates()[0]).toBe(-105);
    expect(geom.getCoordinates()[1]).toBe(40);
  });

  it('should return undefined when pairs for point are incomplete', function() {
    var el = parse('<point>40 -105 50</point>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
  });

  it('should return undefined when points do not contain adequate coordinate pairs',␣
↪function() {
    var el = parse('<point></point>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
  });

  it('should not parse nonsense', function() {
    var el = parse('<point>10 yang</point>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);

    el = parse('<point>ying 10</point>').firstElementChild;
    geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
  });

  it('should parse lines correctly', function() {
    var el = parse('<line>40 100 50 110</line>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom instanceof ol.geom.LineString).toBe(true);
    expect(geom.getCoordinates()[0][0]).toBe(100);
    expect(geom.getCoordinates()[0][1]).toBe(40);
    expect(geom.getCoordinates()[1][0]).toBe(110);
    expect(geom.getCoordinates()[1][1]).toBe(50);
  });

  it('should return undefined when pairs for lines are incomplete', function() {
    var el = parse('<line>40 100 50 110 60</line>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
```

(continues on next page)

(continued from previous page)

```
  });

  it('should return undefined when lines do not contain adequate coordinate pairs',␣
↪function() {
    var el = parse('<line>40 100</line>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
  });

  it('should parse polygons correctly', function() {
    var el = parse('<polygon>40 100 50 110 60 100</polygon>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom instanceof ol.geom.Polygon).toBe(true);
    expect(geom.getCoordinates()[0][0][0]).toBe(100);
    expect(geom.getCoordinates()[0][0][1]).toBe(40);
    expect(geom.getCoordinates()[0][1][0]).toBe(110);
    expect(geom.getCoordinates()[0][1][1]).toBe(50);
    expect(geom.getCoordinates()[0][2][0]).toBe(100);
    expect(geom.getCoordinates()[0][2][1]).toBe(60);
  });

  it('should return undefined when pairs for polygons are incomplete', function() {
    var el = parse('<polygon>40 100 50 110 60 100 70</polygon>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
  });

  it('should return undefined when polygons do not contain adequate coordinate pairs',
↪ function() {
    var el = parse('<polygon>40 100 50 110</polygon>').firstElementChild;
    var geom = GeoRSSParser.parseGeometry(el);
    expect(geom).toBe(undefined);
  });

  it('should return undefined for incorrect tag names', function() {
    var el = parse('<something>is wrong here</something>').firstElementChild;
    expect(GeoRSSParser.parseGeometry(el)).toBe(undefined);
  });

  it('should parse GeoRSS feeds', function() {
    var p = new GeoRSSParser();

    var feed = '<?xml version="1.0" encoding="utf-8"?>' +
      '<feed xmlns="http://www.w3.org/2005/Atom" xmlns:georss="http://www.georss.org/
↪georss">' +
        '<title>Earthquakes</title>' +
        '<subtitle>International earthquake observation labs</subtitle>' +
        '<link href="http://example.org/"/>' +
        '<updated>2005-12-13T18:30:02Z</updated>' +
        '<author>' +
           '<name>Dr. Thaddeus Remor</name>' +
           '<email>tremor@quakelab.edu</email>' +
        '</author>' +
        '<id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>' +
        '<entry>' +
           '<title>M 3.2, Mona Passage</title>' +
           '<link href="http://example.org/2005/09/09/atom01"/>' +
```

(continues on next page)

(continued from previous page)

```
            '<id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>' +
            '<updated>2005-08-17T07:02:32Z</updated>' +
            '<summary>We just had a big one.</summary>' +
            '<georss:point>45.256 -71.92</georss:point>' +
            // we want to make sure this is properly ignored for now, we'll leave it␣
↪as an
            // exercise for the user to implement it
            '<georss:elev>-19372</georss:elev>' +
          '</entry>' +
      '</feed>';

    p.setSource(feed);
    expect(p.hasNext()).toBe(true);

    var source = parse(feed);
    p.setSource(source);
    expect(p.hasNext()).toBe(true);

    var feature = p.parseNext();

    expect(p.hasNext()).toBe(false);
    expect(feature instanceof ol.Feature).toBe(true);
    expect(feature.get('title')).toBe('M 3.2, Mona Passage');
    expect(feature.get('link')).toBe('http://example.org/2005/09/09/atom01');
    expect(feature.get('id')).toBe('urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a');
    expect(feature.get('updated')).toBe('2005-08-17T07:02:32Z');
    expect(feature.get('summary')).toBe('We just had a big one.');
    expect(feature.getGeometry() instanceof ol.geom.Point).toBe(true);

    p.cleanup();
    expect(p.hasNext()).toBe(false);
    expect(p.document).toBe(null);
    expect(p.entries).toBe(null);
    expect(p.nextIndex).toBe(0);
  });

  it('should not use other potential sources', function() {
    var p = new GeoRSSParser();
    p.setSource({something: true});
    expect(p.document).toBe(null);
    expect(p.hasNext()).toBe(false);
  });
});
```

There. Now we can fully test our parser with `yarn test`.

6.1.3 Layer Config

Now that we have a parser, we can hook that up to a layer. OpenSphere automatically configures layers through registered classes dubbed "layer configs". Given a JSON object like:

```
{
  "type": "georss"
}
```

OpenSphere looks up the registered layer config class for the type `georss`, instantiates it, and passes it that JSON

object to create the layer.

Let's create that class.

Listing 5: src/plugin/georss/georsslayerconfig.js

```
goog.declareModuleId('plugin.georss.GeoRSSLayerConfig');

import AbstractDataSourceLayerConfig from 'opensphere/src/os/layer/config/
↪abstractdatasourcelayerconfig.js';
import GeoRSSParser from './georssparser.js';

/**
 * GeoRSS layer config.
 */
export default class GeoRSSLayerConfig extends AbstractDataSourceLayerConfig {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  /**
   * @inheritDoc
   */
  getParser(options) {
    return new GeoRSSParser();
  }
}
```

The parent class, `os.layer.config.AbstractDataSourceLayerConfig` handles most of the heavy lifting and common key/value pairs like `title`, `url`, `description`, etc. All we have to do is pass it a parser.

And, since we are good developers, here is a test for it.

Listing 6: test/plugin/georss/georsslayerconfig.test.js

```
goog.require('plugin.georss.GeoRSSLayerConfig');
goog.require('plugin.georss.GeoRSSParser');

describe('plugin.georss.GeoRSSLayerConfig', function() {
  const {default: GeoRSSLayerConfig} = goog.module.get('plugin.georss.
↪GeoRSSLayerConfig');
  const {default: GeoRSSParser} = goog.module.get('plugin.georss.GeoRSSParser');

  it('should return a GeoRSS parser', function() {
    var config = new GeoRSSLayerConfig();
    expect(config.getParser() instanceof GeoRSSParser).toBe(true);
  });
});
```

Now that we have that tested, we need to modify our plugin and register the layer config:

Listing 7: src/plugin/georss/georssplugin.js

```
goog.declareModuleId('plugin.georss.GeoRSSPlugin');
```

(continues on next page)

(continued from previous page)

```

import LayerConfigManager from 'opensphere/src/os/layer/config/layerconfigmanager.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import {ID} from './georss.js';
import GeoRSSLayerConfig from './georsslayerconfig.js';

/**
 * Provides support for the GeoRSS format.
 */
export default class GeoRSSPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const lcm = LayerConfigManager.getInstance();
    lcm.registerLayerConfig(ID, GeoRSSLayerConfig);
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new GeoRSSPlugin());
```

Running `yarn build` and viewing the debug instance of the application should allow you to drop this in the console and have a GeoRSS layer:

```
const {default: CommandProcessor} = goog.module.get('os.command.CommandProcessor');
const {default: LayerAdd} = goog.module.get('os.command.LayerAdd');
CommandProcessor.getInstance().addCommand(
  new LayerAdd({
    type: 'georss',
    id: 'georss-test',
    title: 'Test GeoRSS Layer',
    url: 'https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_day.atom',
    color: 'FFFF00'
  })
);
```

Cool! But unfortunately this layer is not saved across sessions and it does not support the OpenSphere file/URL import flow. We will do that next!

6.1.4 Content Type Detection

The first thing we need to do for a file type is to detect the file type given a generic file. This is an XML format, so we can extend a generic XML content type detection function from OpenSphere.

Listing 8: `src/plugin/georss/mime.js`:

```
goog.declareModuleId('plugin.georss.mime');

import {register} from 'opensphere/src/os/file/mime.js';
import {TYPE as XMLTYPE, createDetect} from 'opensphere/src/os/file/mime/xml.js';

/**
 * @type {string}
 */
export const TYPE = 'application/rss+xml+geo';

register(
    // the type for this detection
    TYPE,
    // os.file.mime.xml provides a function that creates a detection function for a
    // given root tag regex and xmlns regex
    createDetect(/^feed$/, /^http:\/\/www.w3.org\/2005\/Atom$/),
    // the priority of this detection. 0 is the default, lower numbers run earlier
    0,
    // the parent type; XML in this case
    XMLTYPE);
```

As always, let's test it.

External Plugins If you are working with an external plugin, you will need to add the following two lines to the files list in `karma.conf.js`:

```
{pattern: resolver.resolveModulePath('chardetng-wasm/dist/es5/chardetng.es5.min.js
↪'), watched: false, included: true, served: true},
{pattern: '../opensphere/.build/xml-lexer.min.js', watched: false, included: true,
↪ served: true},
```

Listing 9: `test/plugin/georss/mime.test.js`

```
goog.require('os.file.File');
goog.require('os.file.mime.mock');
goog.require('plugin.georss.mime');

describe('plugin.georss.mime', function() {
  const {default: OSFile} = goog.module.get('os.file.File');
  const {testNo, testYes} = goog.module.get('os.file.mime.mock');
  const {TYPE} = goog.module.get('plugin.georss.mime');

  it('should detect Atom feeds as GeoRSS', function() {
    var feed = '<?xml version="1.0" encoding="utf-8"?>' +
      '<feed xmlns="http://www.w3.org/2005/Atom"/>';

    var buffer = new TextEncoder().encode(feed).buffer;

    // pretend this came from a file
    var file = new OSFile();
    file.setFileName('something.georss');
    file.setUrl(file.getFileName());

    var testFunc = testYes(TYPE);
    testFunc(buffer, file);
```

(continues on next page)

(continued from previous page)

```
  });

  it('should not detect other XML as GeoRSS', function() {
    var xml = '<?xml version="1.0" encoding="utf-8"?><something xmlns="http://
↪something.com/schema"/>';
    var buffer = new TextEncoder().encode(xml).buffer;

    // pretend this came from a file
    var file = new OSFile();
    file.setFileName('something.xml');
    file.setUrl(file.getFileName());

    var testFunc = testNo(TYPE);
    testFunc(buffer, file);
  });

  it('should register itself with mime detection', function() {
    var chain = os.file.mime.getTypeChain(TYPE).join(', ');
    expect(chain).toBe('application/octet-stream, text/plain, text/xml, ' + TYPE);
  });
});
```

Run `yarn test` to try that out.

Now we will have our plugin import our mime package.

Listing 10: `src/plugin/georss/georssplugin.js`

```
goog.declareModuleId('plugin.georss.GeoRSSPlugin');

import './mime.js';

import LayerConfigManager from 'opensphere/src/os/layer/config/layerconfigmanager.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import {ID} from './georss.js';
import GeoRSSLayerConfig from './georsslayerconfig.js';

/**
 * Provides support for the GeoRSS format.
 */
export default class GeoRSSPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
```

(continues on next page)

(continued from previous page)

```
    const lcm = LayerConfigManager.getInstance();
    lcm.registerLayerConfig(ID, GeoRSSLayerConfig);
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new GeoRSSPlugin());
```

Save and run the build. You should now be able to import any atom feed (assuming the remote server has CORS configured; download it and import it as a file otherwise) into OpenSphere! Once it loads, it will complain that it does not have an import UI registered for 'application/rss+xml+geo', which is fine for now.

6.1.5 Import UI Launcher

Import UI Launchers are small classes that configure and launch a UI to let the user set options on the layer. Minimally this should let the user set the layer title and adjust the color.

Listing 11: `src/plugin/georss/georssimportui.js`:

```
goog.declareModuleId('plugin.georss.GeoRSSImportUI');

import FileParserConfig from 'opensphere/src/os/parse/fileparserconfig.js';
import FileImportUI from 'opensphere/src/os/ui/im/fileimportui.js';


/**
 * GeoRSS import UI.
 */
export default class GeoRSSImportUI extends FileImportUI {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  /**
   * @inheritDoc
   */
  getTitle() {
    return 'GeoRSS';
  }

  /**
   * @inheritDoc
   */
  launchUI(file, opt_config) {
    super.launchUI(file, opt_config);

    const config = new FileParserConfig();

    // if an existing config was provided, merge it in
    if (opt_config) {
      this.mergeConfig(opt_config, config);
    }
```

(continues on next page)

(continued from previous page)

```

    config['file'] = file;
    config['title'] = file.getFileName();

    // our config is all set up but we have no UI to launch yet!
  }
}
```

Now we will register our launcher in the plugin.

Listing 12: `src/plugin/georss/georssplugin.js`:

```
goog.declareModuleId('plugin.georss.GeoRSSPlugin');

import LayerConfigManager from 'opensphere/src/os/layer/config/layerconfigmanager.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import ImportManager from 'opensphere/src/os/ui/im/importmanager.js';

import {ID} from './georss.js';
import GeoRSSImportUI from './georssimportui.js';
import GeoRSSLayerConfig from './georsslayerconfig.js';
import {TYPE} from './mime.js';

/**
 * Provides support for the GeoRSS format.
 */
export default class GeoRSSPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const lcm = LayerConfigManager.getInstance();
    lcm.registerLayerConfig(ID, GeoRSSLayerConfig);

    const im = ImportManager.getInstance();
    im.registerImportDetails('GeoRSS', true);
    im.registerImportUI(TYPE, new GeoRSSImportUI());
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new GeoRSSPlugin());
```

Run the build. This gets rid of the error, but our launcher does not launch anything! Let's fix that.

6.1.6 Data Provider

In OpenSphere, the Add Data window (in its default Source view) shows a tree that is essentially Data Providers (root nodes) and Data Descriptors (leaf nodes). We want to add a Data Provider for our GeoRSS file type.

Listing 13: `src/plugin/georss/georssprovider.js`

```
goog.declareModuleId('plugin.georss.GeoRSSProvider');

import FileProvider from 'opensphere/src/os/data/fileprovider.js';
import {ID} from './georss.js';


/**
 * GeoRSS file provider.
 */
export default class GeoRSSProvider extends FileProvider {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  /**
   * @inheritDoc
   */
  configure(config) {
    super.configure(config);
    this.setId(ID);
    this.setLabel('GeoRSS Files');
  }

  /**
   * Get the global instance.
   * @return {!GeoRSSProvider}
   * @export
   */
  static getInstance() {
    if (!instance) {
      instance = new GeoRSSProvider();
    }

    return instance;
  }
}

/**
 * Global instance.
 * @type {GeoRSSProvider|undefined}
 */
let instance;
```

Here is a quick test for it:

Listing 14: test/plugin/georss/georssprovider.test.js

```
goog.require('plugin.georss.GeoRSSProvider');

describe('plugin.georss.GeoRSSProvider', function() {
  const {default: GeoRSSProvider} = goog.module.get('plugin.georss.GeoRSSProvider');

  it('should configure properly', function() {
    const p = new GeoRSSProvider();

    p.configure({
      type: 'georss'
    });

    expect(p.getId()).toBe('georss');
    expect(p.getLabel()).toBe('GeoRSS Files');
  });
});
```

Run `yarn test` to ensure that works. Now let's register it in the plugin.

Listing 15: src/plugin/georss/georssplugin.js

```
goog.declareModuleId('plugin.georss.GeoRSSPlugin');

import DataManager from 'opensphere/src/os/data/datamanager.js';
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import LayerConfigManager from 'opensphere/src/os/layer/config/layerconfigmanager.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import ImportManager from 'opensphere/src/os/ui/im/importmanager.js';

import {ID} from './georss.js';
import GeoRSSImportUI from './georssimportui.js';
import GeoRSSLayerConfig from './georsslayerconfig.js';
import GeoRSSProvider from './georssprovider.js';
import {TYPE} from './mime.js';

/**
 * Provides support for the GeoRSS format.
 */
export default class GeoRSSPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
```

(continues on next page)

(continued from previous page)

```
    const lcm = LayerConfigManager.getInstance();
    lcm.registerLayerConfig(ID, GeoRSSLayerConfig);

    const im = ImportManager.getInstance();
    im.registerImportDetails('GeoRSS', true);
    im.registerImportUI(TYPE, new GeoRSSImportUI());

    // register georss provider type
    const dm = DataManager.getInstance();
    const title = 'GeoRSS Layers';
    dm.registerProviderType(new ProviderEntry(
        ID, // the type
        GeoRSSProvider, // the class
        title, // the title
        title // the description
    ));
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new GeoRSSPlugin());
```

That registers it, but providers are only instantiated if they exist in config (added by either the admin or user in Settings > Data Servers). Let's add some default config. This file is found by looking at `package.json` `build.config`'s values. In OpenSphere it happens to be `config/settings.json`, however, your project could define a different one (or several config files).

This blurb needs to be in your config.

```
{
  "admin": {
    "providers": {
      "georss": {
        "type": "georss"
      }
    }
  }
}
```

Similarly to layer configs, providers are instantiated by type. Pull up the debug instance of OpenSphere and drop this in the console to ensure the new provider is there.

```
const {default: DataManager} = goog.module.get('os.data.DataManager');
const {default: GeoRSSProvider} = goog.module.get('plugin.georss.GeoRSSProvider');
DataManager.getInstance().getProviderRoot().getChildren().filter(
  p => p instanceof GeoRSSProvider)
```

If it is not there, you may have a cache issue with your settings file. You can either navigate to the settings file directly and give it a hard refresh (ctrl/cmd+shift+r in Chrome), clear your full cache, or use the Dev Tools to disable the cache.

6.1.7 Data Descriptor

Now that we have a data provider, we need to create the item that becomes the leaf node in that tree.

Listing 16: src/plugin/georss/georssdescriptor.js

```
goog.declareModuleId('plugin.georss.GeoRSSDescriptor');

import FileDescriptor from 'opensphere/src/os/data/filedescriptor.js';
import LayerType from 'opensphere/src/os/layer/layertype.js';
import ColorControlType from 'opensphere/src/os/ui/colorcontroltype.js';
import ControlType from 'opensphere/src/os/ui/controltype.js';

import {ID} from './georss.js';
import GeoRSSProvider from './georssprovider.js';

const {default: FileParserConfig} = goog.requireType('os.parse.FileParserConfig');


/**
 * GeoRSS file descriptor.
 */
export default class GeoRSSDescriptor extends FileDescriptor {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.descriptorType = ID;
  }

  /**
   * @inheritDoc
   */
  getType() {
    return LayerType.FEATURES;
  }

  /**
   * @inheritDoc
   */
  getLayerOptions() {
    var options = super.getLayerOptions();
    options['type'] = ID;

    // allow resetting the layer color to the default
    options[ControlType.COLOR] = ColorControlType.PICKER_RESET;
    return options;
  }
}

/**
 * Creates a new descriptor from a parser configuration.
 * @param {!FileParserConfig} config
 * @return {!GeoRSSDescriptor}
 */
export const createFromConfig = (config) => {
  const provider = GeoRSSProvider.getInstance();
  const descriptor = new GeoRSSDescriptor();
  FileDescriptor.createFromConfig(descriptor, provider, config);
  return descriptor;
```

(continues on next page)

(continued from previous page)

```
};
```

Let's walk through that real quick. `getType` reports `FEATURES` because we are loading vector data. `getLayerOptions` returns the JSON object that will be fed to the layer config class that we created earlier. The last two static methods assist with creating and updating a descriptor from the model produced by the import UI.

As always, here's the test.

Listing 17: `test/plugin/georss/georssdescriptor.test.js`

```
goog.require('os.file.File');
goog.require('os.layer.LayerType');
goog.require('os.parse.FileParserConfig');
goog.require('os.ui.ColorControlType');
goog.require('os.ui.ControlType');
goog.require('plugin.georss');
goog.require('plugin.georss.GeoRSSDescriptor');
goog.require('plugin.georss.GeoRSSProvider');

describe('plugin.georss.GeoRSSDescriptor', function() {
  const {default: OSFile} = goog.module.get('os.file.File');
  const {default: LayerType} = goog.module.get('os.layer.LayerType');
  const {default: ColorControlType} = goog.module.get('os.ui.ColorControlType');
  const {default: ControlType} = goog.module.get('os.ui.ControlType');
  const {default: FileParserConfig} = goog.module.get('os.parse.FileParserConfig');

  const {ID} = goog.module.get('plugin.georss');
  const {default: GeoRSSDescriptor, createFromConfig} = goog.module.get('plugin.
↪georss.GeoRSSDescriptor');
  const {default: GeoRSSProvider} = goog.module.get('plugin.georss.GeoRSSProvider');

  it('should report the correct type', function() {
    var d = new GeoRSSDescriptor();
    expect(d.getType()).toBe(LayerType.FEATURES);
  });

  it('should produce the correct layer options', function() {
    var d = new GeoRSSDescriptor();
    var opts = d.getLayerOptions();
    expect(opts.type).toBe(ID);
    expect(opts[ControlType.COLOR]).toBe(ColorControlType.PICKER_RESET);
  });

  it('should create a descriptor from a file parser config', function() {
    var file = new OSFile();
    file.setUrl('http://localhost/doesnotexist.georss');

    // default config
    var config = new FileParserConfig(file);
    var d = createFromConfig(config);

    expect(d.getId()).toBeTruthy();
    expect(d.getProvider()).toBe(GeoRSSProvider.getInstance().getLabel());
    expect(d.getUrl()).toBe(file.getUrl());
    expect(d.getColor()).toBeTruthy();

```

(continues on next page)

(continued from previous page)

```
    // edited config
    var config = new FileParserConfig(file);
    config['tags'] = 'one, two\t, three';

    d = createFromConfig(config);
    expect(d.getId()).toBeTruthy();
    expect(d.getProvider()).toBe(GeoRSSProvider.getInstance().getLabel());
    expect(d.getUrl()).toBe(file.getUrl());
    expect(d.getColor()).toBeTruthy();
    expect(d.getTags()).toContain('one');
    expect(d.getTags()).toContain('two');
    expect(d.getTags()).toContain('three');
  });
});
```

Now let's register it in our plugin.

Listing 18: `src/plugin/georss/georssplugin.js`

```
goog.declareModuleId('plugin.georss.GeoRSSPlugin');

import DataManager from 'opensphere/src/os/data/datamanager.js';
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import LayerConfigManager from 'opensphere/src/os/layer/config/layerconfigmanager.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import ImportManager from 'opensphere/src/os/ui/im/importmanager.js';

import {ID} from './georss.js';
import GeoRSSDescriptor from './georssdescriptor.js';
import GeoRSSImportUI from './georssimportui.js';
import GeoRSSLayerConfig from './georsslayerconfig.js';
import GeoRSSProvider from './georssprovider.js';
import {TYPE} from './mime.js';

/**
 * Provides support for the GeoRSS format.
 */
export default class GeoRSSPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const lcm = LayerConfigManager.getInstance();
    lcm.registerLayerConfig(ID, GeoRSSLayerConfig);

    const im = ImportManager.getInstance();
```

(continues on next page)

(continued from previous page)

```
    im.registerImportDetails('GeoRSS', true);
    im.registerImportUI(TYPE, new GeoRSSImportUI());

    // register georss provider type
    const dm = DataManager.getInstance();
    const title = 'GeoRSS Layers';
    dm.registerProviderType(new ProviderEntry(
        ID, // the type
        GeoRSSProvider, // the class
        title, // the title
        title // the description
    ));

    // register the geojson descriptor type
    dm.registerDescriptorType(ID, GeoRSSDescriptor);
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new GeoRSSPlugin());
```

We now have everything we need to get an entry into the Add Data window. So let's do that! Our UI launcher currently does not launch a UI (and it still won't), but it could just save a new descriptor directly without any user input at all. Here's how you hook that up:

Listing 19: `src/plugin/georss/georssimportui.js`

```
goog.declareModuleId('plugin.georss.GeoRSSImportUI');

import DataManager from 'opensphere/src/os/data/datamanager.js';
import DescriptorEvent from 'opensphere/src/os/data/descriptorevent.js';
import DescriptorEventType from 'opensphere/src/os/data/descriptoreventtype.js';
import * as Dispatcher from 'opensphere/src/os/dispatcher.js';
import FileParserConfig from 'opensphere/src/os/parse/fileparserconfig.js';
import FileImportUI from 'opensphere/src/os/ui/im/fileimportui.js';

import {createFromConfig} from './georssdescriptor.js';
import GeoRSSProvider from './georssprovider.js';


/**
 * GeoRSS import UI.
 */
export default class GeoRSSImportUI extends FileImportUI {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  /**
   * @inheritDoc
   */
  getTitle() {
    return 'GeoRSS';
```

(continues on next page)

(continued from previous page)

```
  }

  /**
   * @inheritDoc
   */
  launchUI(file, opt_config) {
    super.launchUI(file, opt_config);

    const config = new FileParserConfig();

    // if an existing config was provided, merge it in
    if (opt_config) {
      this.mergeConfig(opt_config, config);
    }

    config['file'] = file;
    config['title'] = file.getFileName();

    const descriptor = createFromConfig(config);

    // add the descriptor to the data manager first
    DataManager.getInstance().addDescriptor(descriptor);

    // followed by the provider
    GeoRSSProvider.getInstance().addDescriptor(descriptor);

    if (descriptor.isActive()) {
      Dispatcher.getInstance().dispatchEvent(new DescriptorEvent(DescriptorEventType.
↪USER_TOGGLED, descriptor));
    }
  }
}
```

This adds the descriptor to the descriptor list, adds it as a child of the provider, and then pretends that the user went ahead and found it in Add Data and turned it on. To try this out:

1. Go to Open File/URL
2. Paste https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_day.atom or another URL in the field there
3. Hit "Next"

Boom! The layer is on the map. But we kinda had that before. Now it will persist through a restart. So refresh the page and bask in your accomplishment! You can also check it out in the Add Data window, as now there will be a GeoRSS Files > 2.5_day.atom entry. Try importing the same URL again. OpenSphere will detect that you already have a descriptor with the same URL and ask you what you would like to do before continuing.

6.1.8 Import UI

For external plugins, you will need to create at least one `goog.define` for your project. This will allow Angular to find your templates properly.

Listing 20: src/plugin/georss/georss.js

```
goog.declareModuleId('plugin.georss');

/**
 * Plugin identifier.
 * @type {string}
 */
export const ID = 'georss';

/**
 * @define {string} The path to this project.
 */
export const ROOT = goog.define('plugin.georss.ROOT', '../opensphere-plugin-georss/');
```

Note that the path should be the relative path from `opensphere` to your project.

Now we will create an Angular directive that will let the user change the title and color of the the layer.

Listing 21: src/plugin/georss/georssimport.js

```
goog.declareModuleId('plugin.georss.GeoRSSImportUI');

import AbstractFileImportCtrl from 'opensphere/src/os/ui/file/ui/abstractfileimport.js
';
import Module from 'opensphere/src/os/ui/module.js';

import {ROOT} from './georss.js';
import {createFromConfig} from './georssdescriptor.js';
import GeoRSSProvider from './georssprovider.js';

const {default: GeoRSSDescriptor} = goog.requireType('plugin.georss.GeoRSSDescriptor
');


/**
 * The GeoRSS import directive
 * @return {angular.Directive}
 */
/* istanbul ignore next */
export const directive = function() {
  return {
    restrict: 'E',
    replace: true,
    scope: true,
    // The plugin.georss.ROOT define used here helps to fix the paths in the debug␣
↪instance
    // vs. the compiled instance. This example assumes that you are creating an␣
↪external
    // plugin. You do not necessarily need a ROOT define per plugin, but rather per␣
↪project
    // so that the OpenSphere build can find the files properly.
    //
    // For an internal plugin, just require os and use os.ROOT.
    templateUrl: ROOT + 'views/plugin/georss/georssimport.html',
    controller: Controller,
    controllerAs: 'georssImport'
```

(continues on next page)

(continued from previous page)

```
  };
};

/**
 * Add the directive to the module
 */
Module.directive('georssimport', [directive]);

/**
 * Controller for the GeoRSS import dialog
 */
export class Controller extends AbstractFileImportCtrl {
  /**
   * Controller for the GeoRSS import dialog
   * @param {!angular.Scope} $scope
   * @param {!angular.JQLite} $element
   * @ngInject
   */
  constructor($scope, $element) {
    super($scope, $element);
    this.formName = 'georssForm';
  }

  /**
   * @inheritDoc
   */
  createDescriptor() {
    var descriptor = null;
    if (this.config['descriptor']) { // existing descriptor, update it
      descriptor = /** @type {!GeoRSSDescriptor} */ (this.config['descriptor']);
      descriptor.updateFromConfig(this.config);
    } else { // this is a new import
      descriptor = createFromConfig(this.config);
    }

    return descriptor;
  }

  /**
   * @inheritDoc
   */
  getProvider() {
    return GeoRSSProvider.getInstance();
  }
}
```

The parent class, once again, does most of the heavy lifting. We do, however, need to provide the template that we referenced for Angular.

Listing 22: `views/plugin/georss/georssimport.html`

```
<div class="d-flex flex-column flex-fill">
  <div class="modal-body">
    <form name="georssForm">

      <div class="d-flex flex-row form-group">
```

(continues on next page)

(continued from previous page)

```
        <label class="col-3 col-form-label text-right">Layer Title</label>
        <div class="col">
          <input class="form-control" type="text" name="title" ng-model="config.title
↪" ng-required="true" ng-maxlength="50" />
          <validation-message target="georssForm.title"></validation-message>
        </div>
      </div>

      <div class="d-flex flex-row form-group">
        <label class="col-3 col-form-label text-right">Description</label>
        <div class="col">
          <input class="form-control" type="text" name="desc" ng-model="config.
↪description" ng-maxlength="2000" placeholder="Add a custom description"
          />
          <validation-message target="georssForm.desc"></validation-message>
        </div>
      </div>

      <div class="d-flex flex-row form-group">
        <label class="col-3 col-form-label text-right">Tags</label>
        <div class="col">
          <input class="form-control" type="text" name="tags" ng-model="config.tags"␣
↪ng-maxlength="2000" placeholder="Tags organize layers: e.g. states, population"
          />
          <validation-message target="georssForm.tags"></validation-message>
        </div>
      </div>

      <div class="d-flex flex-row form-group">
        <label class="col-3 col-form-label text-right">Color</label>
        <div class="col">
          <colorpicker name="color" class="no-text" color="config.color">
        </div>
      </div>

    </form>
  </div>

  <div class="modal-footer">
    <button class="btn btn-primary" ng-click="georssImport.accept()" ng-disabled=
↪"georssForm.$invalid" title="Import the file">
      <i class="fa fa-check"></i>
      OK
    </button>
    <button class="btn btn-secondary" ng-click="georssImport.cancel()" title="Cancel␣
↪file import">
      <i class="fa fa-ban"></i>
      Cancel
    </button>
  </div>

</div>
```

Cool. Now let's undo our launcher changes from the last step and make it look like this:

Listing 23: src/plugin/georss/georssimportui.js

```
goog.declareModuleId('plugin.georss.GeoRSSImportUI');

import './georssimport.js';

import FileParserConfig from 'opensphere/src/os/parse/fileparserconfig.js';
import FileImportUI from 'opensphere/src/os/ui/im/fileimportui.js';
import {create} from 'opensphere/src/os/ui/window.js';


/**
 * GeoRSS import UI.
 */
export default class GeoRSSImportUI extends FileImportUI {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  // Let's be honest, testing getters like this is pedantic. Let's ignore it
  // this time.
  /* istanbul ignore next */
  /**
   * @inheritDoc
   */
  getTitle() {
    return 'GeoRSS';
  }

  // TODO: This function doesn't do much yet, after it does, let's test the
  // finished product.
  /* istanbul ignore next */
  /**
   * @inheritDoc
   */
  launchUI(file, opt_config) {
    super.launchUI(file, opt_config);

    const config = new FileParserConfig();

    // if an existing config was provided, merge it in
    if (opt_config) {
      this.mergeConfig(opt_config, config);
    }

    config['file'] = file;
    config['title'] = file.getFileName();

    const scopeOptions = {
      'config': config
    };
    const windowOptions = {
      'label': 'Import GeoRSS',
      'icon': 'fa fa-file-text',
```

(continues on next page)

(continued from previous page)

```
      'x': 'center',
      'y': 'center',
      'width': 350,
      'min-width': 350,
      'max-width': 600,
      'height': 'auto',
      'modal': true,
      'show-close': true
    };
    const template = '<georssimport></georssimport>';
    create(windowOptions, template, undefined, undefined, undefined, scopeOptions);
  }
}
```

To test that Import UI, we need a few tests:

Listing 24: `test/plugin/georss/georssimportui.test.js`

```
goog.require('os.file');
goog.require('plugin.georss.GeoRSSImportUI');

describe('plugin.georss.GeoRSSImportUI', function() {
  const {createFromContent} = goog.module.get('os.file');
  const {default: GeoRSSImportUI} = goog.module.get('plugin.georss.GeoRSSImportUI');

  const formSelector = 'form[name="georssForm"]';

  it('should have the proper title', function() {
    var importui = new GeoRSSImportUI();
    expect(importui.getTitle()).toBe('GeoRSS');
  });

  it('should launch an import UI with an empty config', function() {
    var importui = new GeoRSSImportUI();
    var file = createFromContent('testname.rss', 'http://www.example.com/testname.rss
↪', undefined, '<?xml version="1.0" encoding="utf-8"?><feed/>');
    spyOn(os.ui.window, 'create');
    importui.launchUI(file, {});
    expect(os.ui.window.create).toHaveBeenCalled();
  });

  it('should launch an import UI with a null config', function() {
    var importui = new GeoRSSImportUI();
    var file = createFromContent('testname.rss', 'http://www.example.com/testname.rss
↪', undefined, '<?xml version="1.0" encoding="utf-8"?><feed/>');
    importui.launchUI(file, undefined);

    waitsFor(() => !!document.querySelector(formSelector), 'import ui to open');

    runs(() => {
      const formEl = document.querySelector(formSelector);
      const scope = $(formEl).scope();
      expect(scope).toBeDefined();
      expect(scope.georssImport).toBeDefined();

```

(continues on next page)

(continued from previous page)

```
      scope.georssImport.cancel();
    });

    waitsFor(() => !document.querySelector(formSelector), 'import ui to close');
  });

  it('should launch an import UI with a config', function() {
    var importui = new GeoRSSImportUI();
    var file = createFromContent('testname.rss', 'http://www.example.com/testname.rss
↪', undefined, '<?xml version="1.0" encoding="utf-8"?><feed/>');
    importui.launchUI(file, {'title': 'other'});

    waitsFor(() => !!document.querySelector(formSelector), 'import ui to open');

    runs(() => {
      const formEl = document.querySelector(formSelector);
      const scope = $(formEl).scope();
      expect(scope).toBeDefined();
      expect(scope.georssImport).toBeDefined();

      scope.georssImport.cancel();
    });

    waitsFor(() => !document.querySelector(formSelector), 'import ui to close');
  });
});
```

Save, build, test, and pull it up.

1. Go to Add Data > GeoRSS Files and delete any entries under there by highlighting and clicking the trash can
2. Import https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_day.atom or your URL again

This time it should launch the UI that we just made. Change the title, description, tags, and or color and hit "OK" to save it.

Most of our import UIs are not quite this simple. Even GeoJSON requires the user to set up time mappings. Our format has the `updated` field which contains a time, so let's get that supported in order for the layer to animate properly.

6.1.9 Time Support

You may have noticed that there is a little clock icon on the layer. The parent class to our descriptor, `os.data.FileDescriptor`, sets the `animate` flag to `true` in our layer options:

```
  getLayerOptions() {
    var options = {};
    options['id'] = this.getId();

    options['animate'] = true; // TODO: add checkbox to toggle this in import UI
    options['color'] = this.getColor();
    options['icon'] = this.getIcon();
    options['shapeName'] = this.getShapeName();
    options['load'] = true;
    options['originalUrl'] = this.getOriginalUrl();
    options['parserConfig'] = this.parserConfig;
    options['provider'] = this.getProvider();
```

(continues on next page)

(continued from previous page)

```
    options['tags'] = this.getTags();
    options['title'] = this.getTitle();
    options['url'] = this.getUrl();
    options['mappings'] = this.getMappings();
    options['detectColumnTypes'] = true;

    return options;
  }
```

That turns on time indexing on the layer. However, nothing is taking the time values from the `<updated>` tags in the GeoRSS entries and turning them into `os.time.TimeInstant` values in the proper place on the feature. Hence every feature is treated as "timeless" even though the source has time indexing enabled. This is where mappings come in.

Import mappings all implement `os.im.mapping.IMapping` and are used to either change fields (e.g. unit conversion) or derive fields (e.g. LAT/LON columns in CSVs to `ol.geom.Point`, adding altitude, etc.). However, the most common mappings of all are time mappings. Import UIs for types such as GeoJSON and CSV generally walk the user through creating these mappings (CSV, being the worst file format of all time for geospatial data, has to basically create every type of mapping possible).

In our case, we have several options.

Reinvent the wheel

We could simply parse the time in our parser since the format is common and only shows up in one place (the `<updated>` tag). KML is similar in this regard as the format and tags for time are defined in the spec. But that really just duplicates code that exists within the mappings.

Define a mapping explicitly

We could define a `os.im.mapping.time.DateTimeMapping` instance ourselves. This could be placed in several different places but the best place is probably the layer config. That would look something like this:

Listing 25: `src/plugin/georss/georsslayerconfig.js`

```
goog.declareModuleId('plugin.georss.GeoRSSLayerConfig');

import DateTimeMapping from 'opensphere/src/os/im/mapping/time/datetimemapping.js';
import TimeType from 'opensphere/src/os/im/mapping/timetype.js';
import AbstractDataSourceLayerConfig from 'opensphere/src/os/layer/config/
↪abstractdatasourcelayerconfig.js';

import GeoRSSParser from './georssparser.js';

/**
 * GeoRSS layer config.
 */
export default class GeoRSSLayerConfig extends AbstractDataSourceLayerConfig {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  /**
   * @inheritDoc
   */
```

(continues on next page)

(continued from previous page)

```
  getParser(options) {
    return new GeoRSSParser();
  }

  /**
   * @inheritDoc
   */
  getImporter(options) {
    const importer = super.getImporter(options);

    // add time mapping to importer
    const timeMapping = new DateTimeMapping(TimeType.INSTANT);
    timeMapping.field = 'updated';
    // there's no need to call timeMapping.setFormat() since the default is what we want

    importer.setExecMappings([timeMapping]);
    return importer;
  }
}
```

Use the auto-detected mappings

Because many import UIs configure mappings, it helps the user if those default to the best value possible. Therefore, OpenSphere includes auto-detection logic with each mapping to make a best guess on what the mapping should be based on the fields and values available in a sample of records from the source. We could simply make use of that like so:

Listing 26: `src/plugin/georss/georsslayerconfig.js`

```
goog.declareModuleId('plugin.georss.GeoRSSLayerConfig');

import DateTimeMapping from 'opensphere/src/os/im/mapping/time/datetimemapping.js';
import AbstractDataSourceLayerConfig from 'opensphere/src/os/layer/config/abstractdatasourcelayerconfig.js';

import GeoRSSParser from './georssparser.js';

/**
 * GeoRSS layer config.
 */
export default class GeoRSSLayerConfig extends AbstractDataSourceLayerConfig {
  /**
   * Constructor.
   */
  constructor() {
    super();
  }

  /**
   * @inheritDoc
   */
  getParser(options) {
    return new GeoRSSParser();
  }

  /**
```

(continues on next page)

(continued from previous page)

```
    * @inheritDoc
    */
  getImporter(options) {
    const importer = super.getImporter(options);

    // Auto detect the time from the fields.
    // This should only be called for mappings not explicitly set by the user in
    // the import UI. See other file types for examples.
    importer.selectAutoMappings([DateTimeMapping.ID]);

    return importer;
  }
}
```

After picking any of those options, fire up the debug instance and load your URL. Open the timeline by clicking the yellow clock icon in the Date Control at the top or choosing Windows > Timeline. If you were using a recent feed, you should see your data for today immediately and you can hit the Play button to animate it. If your data is older, you can zoom/pan to it and draw (or explicitly set) a new Load or Animation range before playing.

To complete testing of the importer, we can extend the tests as shown below:

Listing 27: `test/plugin/georss/georsslayerconfig.test.js`

```
goog.require('os.im.FeatureImporter');
goog.require('plugin.georss.GeoRSSLayerConfig');
goog.require('plugin.georss.GeoRSSParser');

describe('plugin.georss.GeoRSSLayerConfig', function() {
  const {default: FeatureImporter} = goog.module.get('os.im.FeatureImporter');
  const {default: GeoRSSLayerConfig} = goog.module.get('plugin.georss.
↪GeoRSSLayerConfig');
  const {default: GeoRSSParser} = goog.module.get('plugin.georss.GeoRSSParser');

  it('should return a GeoRSS parser', function() {
    var config = new GeoRSSLayerConfig();
    expect(config.getParser() instanceof GeoRSSParser).toBe(true);
  });

  it('should return a GeoRSS importer', function() {
    var config = new GeoRSSLayerConfig();
    expect(config.getImporter('http://www.example.com/testname.rss', {}) instanceof␣
↪FeatureImporter).toBe(true);
  });
});
```

That concludes the File Type Plugin Guide! If you have any further questions or requests for new guides, please feel free to use our GitHub issues. Also, remember to check out the core plugins as examples for other things you can do!.

6.2 Server Plugin Guide

Here we walk through creating a plugin for a new server or provider type: Tileserver-GL. There are other examples of these in OpenSphere, such as OGC (WMS/WFS) and ArcGIS.

With any server type, it is important to have an instance we can hit for testing. Follow the instructions in the Tileserver-

GL README (download a datafile and fire up the Docker container). If you have installed httpd or nginx, you may want to pick a different port mapping from those instructions so that the ports do not conflict. This guide assumes port 8081.

After that is done, you should be able to hit `http://localhost:<port>/index.json`. This is the file that our server type will read.

If you wish for this to be an external, separately released plugin, then fork opensphere-plugin-example and follow the instructions in its readme as a starting point.

If you wish for this to be a core plugin included with OpenSphere then simply begin adding your plugin code to `src/plugin/yourplugin`. However, if you are serious about getting your plugin included to the core project, please create an issue for it so we can discuss it.

6.2.1 Plugin

Add a constant for the plugin id.

Listing 28: `src/plugin/tileserver/index.js`

```
goog.declareModuleId('plugin.tileserver');

/**
 * @type {string}
 */
export const ID = 'tileserver';
```

Add a basic plugin class.

Listing 29: `src/plugin/tileserver/tileserverplugin.js`

```
goog.declareModuleId('plugin.tileserver.TileserverPlugin');

import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import {ID} from './index.js';

/**
 * Provides Tileserver support
 */
export default class TileserverPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    // our plugin doesn't do anything yet
```

(continues on next page)

(continued from previous page)

```
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TileserverPlugin());
```

Internal Plugins If creating an internal plugin, ensure that `mainctrl.js` `goog.require`'s your plugin, or optionally, the external plugin method will also work.

External Plugins If creating an external plugin, ensure that `package.json` `build.gcc.entry_point` has `goog:plugin.tileserver.TileserverPlugin` in its list.

Run `yarn build` in OpenSphere (not in your plugin if it is external). It should build just fine but it does not do anything yet.

Just for good measure, let's test it.

Listing 30: `test/plugin/tileserver/tileserverplugin.test.js`

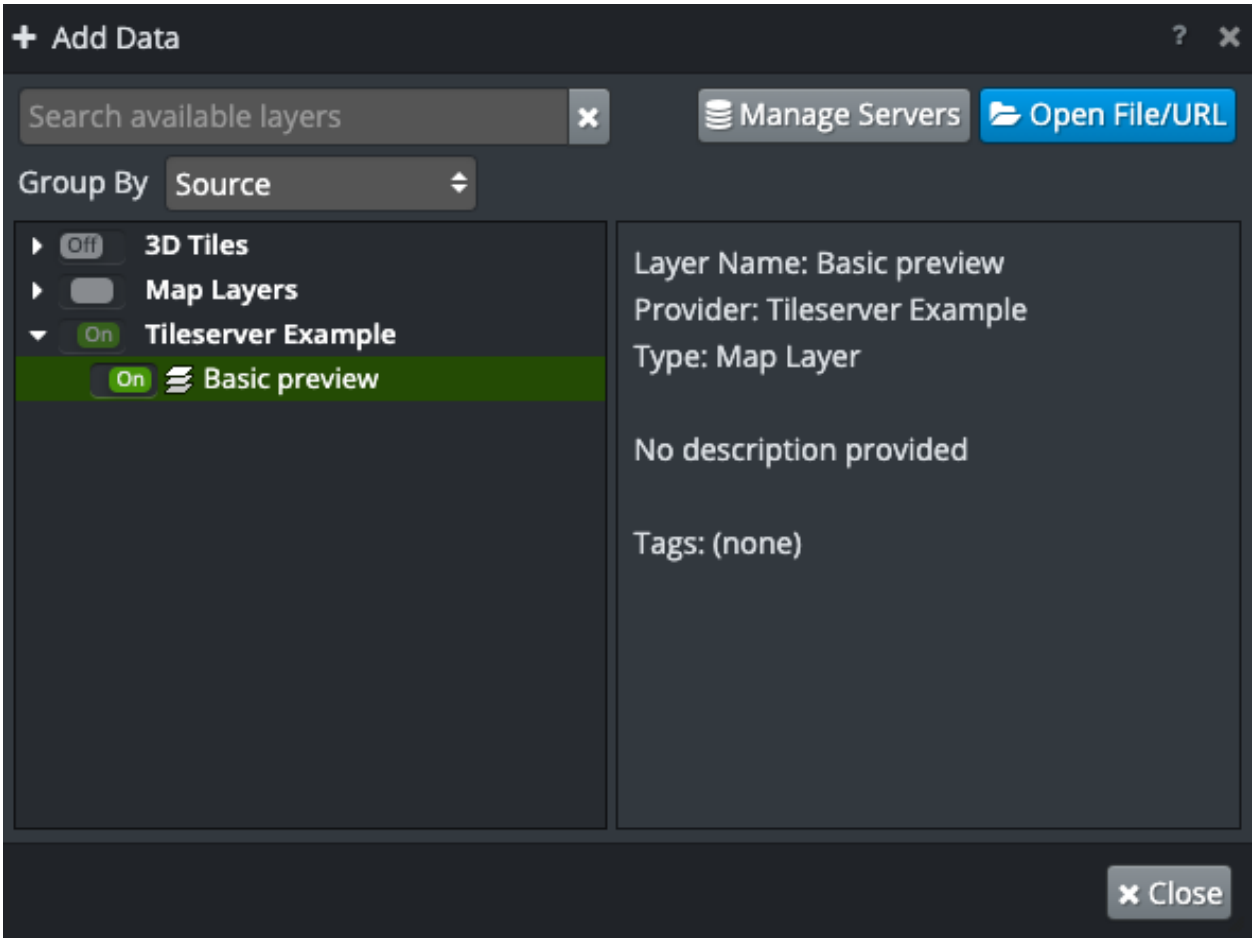
```
// os.mock sets up a bunch of basic opensphere APIs, like settings, which is
// used in our example plugin
goog.require('os.mock');
goog.require('plugin.tileserver.TileserverPlugin');

describe('plugin.tileserver.TileserverPlugin', function() {
  const {default: TileserverPlugin} = goog.module.get('plugin.tileserver.
↪TileserverPlugin');

  it('should have the proper ID', function() {
    expect(new TileserverPlugin().id).toBe('tileserver');
  });

  it('should not throw an error', function() {
    const fn = function() {
      const p = new TileserverPlugin();
      p.init();
    };

    expect(fn).not.toThrow();
  });
});
```

Run `yarn test` in your plugin project if it is external (not in OpenSphere) to run its tests. For internal plugins, no directory change is necessary.

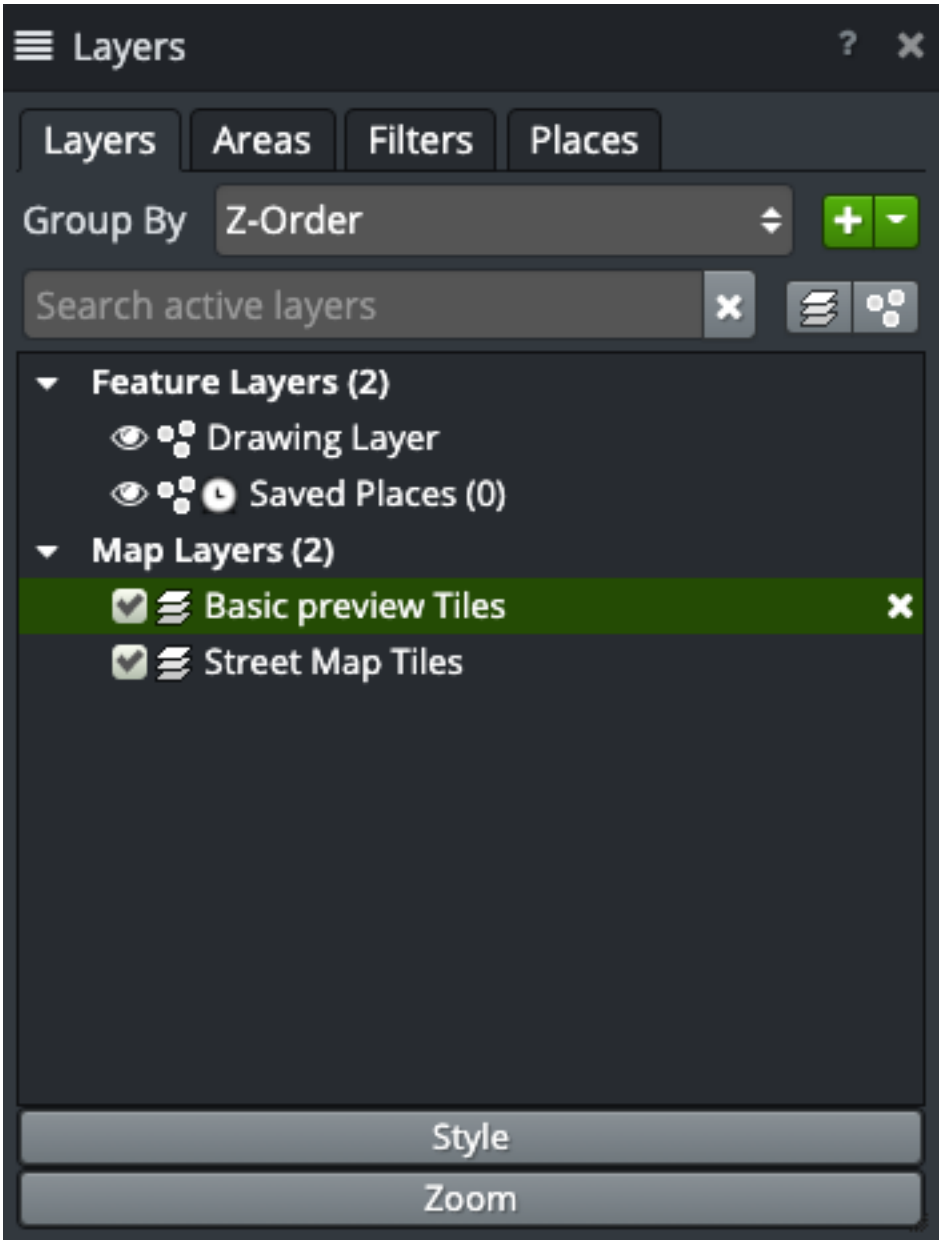
6.2.2 Data Provider

Here we will add a provider class that reads the `/index.json` file from Tileserver and produces layers from it.

A Provider is represented visually by the root nodes in the Add Data window (in its default view). Its goal is to query its server and produce leaf nodes containing Data Descriptors. These leaf nodes are represented visually by the nodes with on/off sliders in that same view, as shown below:



When a layer is turned on, it appears as a node in the layers menu, as shown below:



While most descriptors activate layers, a custom descriptor could activate or launch anything.

First let's just get the JSON loaded.

Listing 31: `src/plugin/tileserver/tileserver.js`

```
goog.declareModuleId('plugin.tileserver.Tileserver');

import Request from 'opensphere/src/os/net/request.js';
import AbstractLoadingServer from 'opensphere/src/os/ui/server/abstractloadingserver.
↪js';

import {ID} from './index.js';

const {default: IDataProvider} = goog.requireType('os.data.IDataProvider');
```

(continues on next page)

(continued from previous page)

```

/**
 * The Tileserver provider
 * @implements {IDataProvider}
 */
export default class Tileserver extends AbstractLoadingServer {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.providerType = ID;
  }

  /**
   * @inheritDoc
   */
  load(opt_ping) {
    super.load(opt_pint);

    // load the JSON
    new Request(this.getUrl()).getPromise().
        then(this.onLoad, this.onError, this).
        thenCatch(this.onError, this);
  }

  /**
   * @param {string} response
   * @protected
   */
  onLoad(response) {
    let layers;

    try {
      layers = JSON.parse(response);
    } catch (e) {
      this.onError('Malformed JSON');
      return;
    }

    console.log('parsed json', layers);
  }

  /**
   * @param {*} e
   * @protected
   */
  onError(e) {
    const msg = Array.isArray(e) ? e.join(' ') : e.toString();
    this.setErrorMessage(msg);
  }
}
```

Here's our test for it. Note that this uses the same Jasmine 1.3 that OpenSphere uses. Newer versions allow for async tests with promises. If you have an external plugin and would like to use a newer or different test library, that is up to you.

Listing 32: test/plugin/tileserver/tileserver.test.js

```
goog.require('goog.Promise');
goog.require('os.net.Request');
goog.require('plugin.tileserver');
goog.require('plugin.tileserver.Tileserver');

describe('plugin.tileserver.Tileserver', function() {
  const Promise = goog.module.get('goog.Promise');
  const {default: Request} = goog.module.get('os.net.Request');

  const {ID} = goog.module.get('plugin.tileserver');
  const {default: Tileserver} = goog.module.get('plugin.tileserver.Tileserver');

  it('should configure properly', function() {
    var p = new Tileserver();
    var conf = {
      type: ID,
      label: 'Test Server',
      url: 'http://localhost/doesnotexist.json'
    };

    p.configure(conf);

    expect(p.getLabel()).toBe(conf.label);
    expect(p.getUrl()).toBe(conf.url);
  });

  it('should load valid JSON', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise resolving
    // to valid JSON
    spyOn(Request.prototype, 'getPromise').andReturn(Promise.resolve('[]'));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onLoad.calls.length;
    });

    runs(function() {
      expect(p.onLoad).toHaveBeenCalled();
      expect(p.onError).not.toHaveBeenCalled();
    });
  });

  it('should error on invalid JSON', function() {
    var p = new Tileserver();
    p.setUrl('/something');
```

(continues on next page)

(continued from previous page)

```

    // we're going to spy on the getPromise method and return a promise resolving
    // to invalid JSON
    spyOn(Request.prototype, 'getPromise').andReturn(Promise.resolve('[wut'));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onLoad.calls.length;
    });

    runs(function() {
      expect(p.onLoad).toHaveBeenCalled();
      expect(p.onError).toHaveBeenCalled();
    });
  });

  it('should error on request error', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise rejecting
    // with errors
    spyOn(Request.prototype, 'getPromise').andReturn(
        // request rejects with arrays of all errors that occurred
        Promise.reject(['something awful happend']));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onError.calls.length;
    });

    runs(function() {
      expect(p.onLoad).not.toHaveBeenCalled();
      expect(p.onError).toHaveBeenCalled();
    });
  });
});
```

Run `yarn test` to see if it tests properly. Now we need to register our provider type in our plugin.

Listing 33: `src/plugin/tileserer/tileserverplugin.js`

```
goog.declareModuleId('plugin.tileserver.TileserverPlugin');

import DataManager from 'opensphere/src/os/data/datamanager.js';
```

(continues on next page)

(continued from previous page)

```
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import Tileserver from './tileserver.js';
import {ID} from './index.js';

/**
 * Provides Tileserver support
 */
export default class TileserverPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const dm = DataManager.getInstance();
    dm.registerProviderType(new ProviderEntry(
        ID, // the type
        Tileserver, // the class
        'Tileserver', // the title
        'Tileserver layers' // the description
    ));
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TileserverPlugin());
```

Lastly, we need to update our config so that the application instantiates a copy of our provider.

Listing 34: `config/settings.json`

```
{
  "admin": {
    "providers": {
      "tileserver-example": {
        "type": "tileserver",
        "label": "Tileserver Example",
        "url": "http://localhost:8081/index.json"
      }
    }
  }
}
```

Run the build in OpenSphere and open the debug instance of the application. You should see a copy of the provider in both the Add Data window and in Settings > Data Servers and you can see our `console.log` statement in the Javascript console. However, our server is just spinning and is not processing that JSON yet. Let's fix that next.

6.2.3 Parsing

Let's have a look at the JSON response from Tileserver.

Listing 35: `/index.json`

```
[
  {
    "tilejson": "2.0.0",
    "name": "Basic preview",
    "attribution": "<a href=\"http://openmaptiles.org/\" target=\"_blank\">&copy;␣
↪OpenMapTiles</a> <a href=\"http://www.openstreetmap.org/about/\" target=\"_blank\">&
↪copy; OpenStreetMap contributors</a>",
    "minzoom": 0,
    "maxzoom": 20,
    "bounds": [8.275, 47.225, 8.8, 47.533],
    "format": "png",
    "type": "baselayer",
    "tiles": ["http://localhost:8081/styles/basic-preview/{z}/{x}/{y}.png"],
    "center": [8.537500000000001, 47.379000000000005, 11]
  },
  {
    "tiles": ["http://localhost:8081/data/v3/{z}/{x}/{y}.pbf"],
    "name": "OpenMapTiles",
    "format": "pbf",
    "basename": "zurich_switzerland.mbtiles",
    "id": "openmaptiles",
    "attribution": "<a href=\"http://openmaptiles.org/\" target=\"_blank\">&copy;␣
↪OpenMapTiles</a> <a href=\"http://www.openstreetmap.org/about/\" target=\"_blank\">&
↪copy; OpenStreetMap contributors</a>",
    "description": "https://openmaptiles.org",
    "minzoom": 0,
    "maxzoom": 14,
    "center": [8.5375, 47.379, 10],
    "bounds": [8.275, 47.225, 8.8, 47.533],
    "version": "3.3",
    "type": "overlay",
    "tilejson": "2.0.0"
```

(continues on next page)

(continued from previous page)

```
  }
]
```

The first layer is an XYZ (see `tiles` template) PNG (`format = 'png'`) layer that is intended as base map imagery (`type = 'baselayer'`). OpenSphere already supports XYZ layers and base map layers, so we can take advantage of a lot of code that already exists.

The second layer is a vector format, which we will ignore in this guide. The first JSON object is very similar to the layer config JSON needed to create XYZ layers. So all we have to do is translate it.

Listing 36: `src/plugin/tileserver/tileserver.js`

```
goog.declareModuleId('plugin.tileserver.Tileserver');

import ConfigDescriptor from 'opensphere/src/os/data/configdescriptor.js';
import DataManager from 'opensphere/src/os/data/datamanager.js';
import {filterFalsey} from 'opensphere/src/os/fn/fn.js';
import {MAX_ZOOM, MIN_ZOOM} from 'opensphere/src/os/map/map.js';
import Request from 'opensphere/src/os/net/request.js';
import {EPSG3857, EPSG4326} from 'opensphere/src/os/proj/proj.js';
import BaseProvider from 'opensphere/src/os/ui/data/baseprovider.js';
import DescriptorNode from 'opensphere/src/os/ui/data/descriptornode.js';
import {createIconSet} from 'opensphere/src/os/ui/icons/index.js';
import IconsSVG from 'opensphere/src/os/ui/iconssvg.js';
import AbstractLoadingServer from 'opensphere/src/os/ui/server/abstractloadingserver.
↪js';
import * as basemap from 'opensphere/src/plugin/basemap/basemap.js';

import {ID} from './index.js';

const {default: IDataProvider} = goog.requireType('os.data.IDataProvider');


/**
 * The Tileserver provider
 * @implements {IDataProvider}
 */
export default class Tileserver extends AbstractLoadingServer {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.providerType = ID;
  }

  /**
   * @inheritDoc
   */
  load(opt_ping) {
    super.load(opt_ping);

    // clear out any children we already have
    this.setChildren(null);

    // load the JSON
    new Request(this.getUrl()).getPromise().
```

(continues on next page)

(continued from previous page)

```
        then(this.onLoad, this.onError, this).
        thenCatch(this.onError, this);
  }

  /**
   * @param {string} response
   * @protected
   */
  onLoad(response) {
    let layers;

    try {
      layers = JSON.parse(response);
    } catch (e) {
      this.onError('Malformed JSON');
      return;
    }

    if (!Array.isArray(layers)) {
      // not sure what we got but it isn't an array of layers
      this.onError('Expected an array of layers but got something else');
      return;
    }

    var children = /** @type {Array<!os.structs.ITreeNode>} */ (
      layers.map(this.toChildNode, this).filter(filterFalsey));
    this.setChildren(children);
    this.finish();
  }

  /**
   * @param {Object<string, *>} layer The layer JSON
   * @return {?DescriptorNode} The child node for the provider
   * @protected
   */
  toChildNode(layer) {
    if (!layer['tilejson']) {
      return null;
    }

    if (!/^(png|jpe?g|gif)$/i.test(layer['format'])) {
      // not our format
      return null;
    }

    var id = this.getId() + BaseProvider.ID_DELIMITER + layer['name'];

    var config = {
      'type': 'XYZ',
      'id': id,
      'title': layer['name'],
      'urls': layer['tiles'],
      'extent': layer['bounds'],
      'extentProjection': EPSG4326,
      'icons': createIconSet(id, [IconsSVG.TILES], [], [255, 255, 255, 1]),
      'projection': EPSG3857,
      'minZoom': Math.max(MIN_ZOOM, layer['minzoom']),
```

(continues on next page)

(continued from previous page)

```
      'maxZoom': Math.min(MAX_ZOOM, layer['maxzoom']),
      'attributions': [layer['attribution']],
      'provider': this.getLabel(),
      // this delays enabling the descriptor on startup until this provider marks it as ready
      'delayUpdateActive': true
    };

    if (layer['type'] === 'baselayer') {
      config['baseType'] = config['type'];
      config['type'] = basemap.ID;
      config['layerType'] = basemap.LAYER_TYPE;
      config['noClear'] = true;
    }

    var descriptor = /** @type {ConfigDescriptor} */ (DataManager.getInstance().getDescriptor(id));
    if (!descriptor) {
      descriptor = new ConfigDescriptor();
    }

    descriptor.setBaseConfig(config);

    // add the descriptor to the data manager
    DataManager.getInstance().addDescriptor(descriptor);

    // mark the descriptor as ready if the user had it enabled previously
    descriptor.updateActiveFromTemp();

    var node = new DescriptorNode();
    node.setDescriptor(descriptor);

    return node;
  }

  /**
   * @param {*} e
   * @protected
   */
  onError(e) {
    var msg = Array.isArray(e) ? e.join(' ') : e.toString();
    this.setErrorMessage(msg);
  }
}
```

Let's test it.

Listing 37: `test/plugin/tileserver/tileserver.test.js`

```
goog.require('goog.Promise');
goog.require('os.data.ConfigDescriptor');
goog.require('os.net.Request');
goog.require('os.ui.data.BaseProvider');
goog.require('plugin.tileserver');
goog.require('plugin.tileserver.Tileserver');
```

(continues on next page)

(continued from previous page)

```

describe('plugin.tileserver.Tileserver', function() {
  const Promise = goog.module.get('goog.Promise');
  const {default: ConfigDescriptor} = goog.module.get('os.data.ConfigDescriptor');
  const {default: Request} = goog.module.get('os.net.Request');
  const {default: BaseProvider} = goog.module.get('os.ui.data.BaseProvider');

  const {ID} = goog.module.get('plugin.tileserver');
  const {default: Tileserver} = goog.module.get('plugin.tileserver.Tileserver');

  it('should configure properly', function() {
    var p = new Tileserver();
    var conf = {
      type: ID,
      label: 'Test Server',
      url: 'http://localhost/doesnotexist.json'
    };

    p.configure(conf);

    expect(p.getLabel()).toBe(conf.label);
    expect(p.getUrl()).toBe(conf.url);
  });

  it('should load valid JSON', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise resolving
    // to valid JSON
    spyOn(Request.prototype, 'getPromise').andReturn(Promise.resolve('[]'));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onLoad.calls.length;
    });

    runs(function() {
      expect(p.onLoad).toHaveBeenCalled();
      expect(p.onError).not.toHaveBeenCalled();
    });
  });

  it('should error on invalid JSON', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise resolving
    // to invalid JSON
    spyOn(Request.prototype, 'getPromise').andReturn(Promise.resolve('[wut'));

```

(continues on next page)

(continued from previous page)

```
    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onLoad.calls.length;
    });

    runs(function() {
      expect(p.onLoad).toHaveBeenCalled();
      expect(p.onError).toHaveBeenCalled();
    });
  });

  it('should error on request error', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise rejecting
    // with errors
    spyOn(Request.prototype, 'getPromise').andReturn(
        // request rejects with arrays of all errors that occurred
        Promise.reject(['something awful happend']));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onError.calls.length;
    });

    runs(function() {
      expect(p.onLoad).not.toHaveBeenCalled();
      expect(p.onError).toHaveBeenCalled();
    });
  });

  it('should ignore JSON that is not an array', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise resolving
    // to valid JSON
    spyOn(Request.prototype, 'getPromise').andReturn(Promise.resolve('{}'));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    runs(function() {
      p.load();
```

(continues on next page)

(continued from previous page)

```
    });

    waitsFor(function() {
      return p.onLoad.calls.length;
    });

    runs(function() {
      expect(p.onLoad).toHaveBeenCalled();
      expect(p.onError).toHaveBeenCalled();
    });
  });

  it('should parse Tileserver JSON', function() {
    var p = new Tileserver();
    p.setUrl('/something');

    // we're going to spy on the getPromise method and return a promise resolving to some Tileserver JSON
    spyOn(Request.prototype, 'getPromise').andReturn(Promise.resolve(JSON.stringify(
        [{
          tilejson: '2.0.0',
          name: 'Klokantech Basic',
          attribution: 'This is a test',
          minzoom: 0,
          maxzoom: 20,
          bounds: [8.275, 47.225, 8.8, 47.533],
          format: 'png',
          type: 'baselayer',
          tiles: ['http://localhost:8081/styles/klokantech-basic/{z}/{x}/{y}.png'],
          center: [8.537500000000001, 47.379000000000005, 11]
        }, {
          tilejson: '2.0.0',
          name: 'Klokantech Basic 2',
          attribution: 'This is a test',
          minzoom: 0,
          maxzoom: 20,
          bounds: [8.275, 47.225, 8.8, 47.533],
          format: 'png',
          tiles: ['http://localhost:8081/styles/klokantech-basic/{z}/{x}/{y}.png'],
          center: [8.537500000000001, 47.379000000000005, 11]
        }, {
          tilejson: '2.0.0',
          format: 'pbf'
        }, {
          something: true
        }]
    )));

    spyOn(p, 'onLoad').andCallThrough();
    spyOn(p, 'onError').andCallThrough();

    // add a config descriptor to the datamanager so that we can test updating on one of the layers
    var id = p.getId() + BaseProvider.ID_DELIMITER + 'Klokantech Basic';
    var descriptor = new ConfigDescriptor();
    descriptor.setBaseConfig({
      id: id
```

(continues on next page)

(continued from previous page)

```
    });
    os.dataManager.addDescriptor(descriptor);

    runs(function() {
      p.load();
    });

    waitsFor(function() {
      return p.onLoad.calls.length;
    });

    runs(function() {
      expect(p.onLoad).toHaveBeenCalled();
      expect(p.onError).not.toHaveBeenCalled();
      expect(p.getChildren().length).toBe(2);
      expect(p.getChildren()[0].getLabel()).toBe('Klokantech Basic');
      expect(p.getChildren()[1].getLabel()).toBe('Klokantech Basic 2');
    });
  });
});
```

`yarn test` should result in a clean test run. Now let's ensure that our descriptor type is registered in our plugin.

Listing 38: `src/plugin/tileserver/tileserverplugin.js`

```
goog.declareModuleId('plugin.tileserver.TileserverPlugin');

import ConfigDescriptor from 'opensphere/src/os/data/configdescriptor.js';
import DataManager from 'opensphere/src/os/data/datamanager.js';
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import Tileserver from './tileserver.js';
import {ID} from './index.js';

/**
 * Provides Tileserver support
 */
export default class TileserverPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const dm = DataManager.getInstance();
    dm.registerProviderType(new ProviderEntry(
```

(continues on next page)

(continued from previous page)

```
        ID, // the type
        Tileserver, // the class
        'Tileserver', // the title
        'Tileserver layers' // the description
    ));

    dm.registerDescriptorType(ID, ConfigDescriptor);
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TileserverPlugin());
```

Since `ConfigDescriptor` is highly resuable, it is possible that several plugins make that same registration. That's fine.

Now run the build and open up the debug instance again. This time the server should complete its loading and show a child node for the parsed layer. Toggling the child should enable the given layer, and the layer should persist on refresh.

We've got the provider working if added by an admin in config, but what about the user? Let's handle that next.

6.2.4 Server Import UI

Server type detection works exactly the same as file type detection. Let's add a detection function for a Tileserver `/index.json` response and register it.

Listing 39: `src/plugin/tileserver/mime.js`

```
goog.declareModuleId('plugin.tileserver.mime');

import {register} from 'opensphere/src/os/file/mime.js';
import {TYPE} from 'opensphere/src/os/file/mime/json.js';
import {ID} from './index.js';

const Promise = goog.require('goog.Promise');
const {default: OSFile} = goog.requireType('os.file.File');


/**
 * @param {ArrayBuffer} buffer
 * @param {OSFile} file
 * @param {*=} opt_context
 * @return {!Promise<*|undefined>}
 */
export const detect = function(buffer, file, opt_context) {
  var retVal;

  // the parent type (JSON) gives the context as the parsed JSON
  // (so far)

  if (opt_context && Array.isArray(opt_context) && opt_context.length && 'tilejson'␣
↪in opt_context[0]) {
    retVal = opt_context;
  }

```

(continues on next page)

(continued from previous page)

```
  return Promise.resolve(retVal);
};

register(
    // for providers, this must be the same as the ProviderEntry ID
    ID,
    detect,
    // the priority; lower numbers run earlier
    0,
    // the parent type
    TYPE);
```

Let's test that.

Listing 40: `test/plugin/tileserver/mime.test.js`

```
goog.require('os.file.File');
goog.require('os.file.mime');
goog.require('os.file.mime.mock');
goog.require('plugin.tileserver.mime');

describe('plugin.tileserver.mime', function() {
  const {default: OSFile} = goog.module.get('os.file.File');
  const {getTypeChain} = goog.module.get('os.file.mime');
  const {testNo, testYes} = goog.module.get('os.file.mime.mock');
  const {ID} = goog.module.get('plugin.tileserver');

  it('should detect tileserver responses', function() {
    var json = '  [{"format":"png","tilejson":"2.0.0"}] ';
    var buffer = new TextEncoder().encode(json);

    var file = new OSFile();
    file.setUrl('http://something.com/index.json');

    var testFunc = testYes(ID);
    testFunc(buffer, file);
  });

  it('should not detect other JSON', function() {
    var json = '  {type: "FeatureCollection"} ';
    var buffer = new TextEncoder().encode(json);

    var file = new OSFile();
    file.setUrl('http://something.com/other.json');

    var testFunc = testNo(ID);
    testFunc(buffer, file);
  });

  it('should register itself with mime detection', function() {
    var chain = getTypeChain(ID).join(', ');
    expect(chain).toBe('application/octet-stream, text/plain, text/xml, ' + ID);
  });
});
```

Now have the plugin import it.

Listing 41: src/plugin/tileserver/tileserverplugin.js

```
goog.declareModuleId('plugin.tileserver.TileserverPlugin');

import './mime.js';

import ConfigDescriptor from 'opensphere/src/os/data/configdescriptor.js';
import DataManager from 'opensphere/src/os/data/datamanager.js';
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

import Tileserver from './tileserver.js';
import {ID} from './index.js';

/**
 * Provides Tileserver support
 */
export default class TileserverPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const dm = DataManager.getInstance();
    dm.registerProviderType(new ProviderEntry(
        ID, // the type
        Tileserver, // the class
        'Tileserver', // the title
        'Tileserver layers' // the description
    ));

    dm.registerDescriptorType(ID, ConfigDescriptor);
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TileserverPlugin());
```

Now we need to make an Angular directive so the user has a form to give the server a title and potentially modify the URL.

Listing 42: src/plugin/tileserver/tileserverimport.js

```
goog.declareModuleId('plugin.tileserver.TileserverImportUI');

import {ROOT} from 'opensphere/src/os/os.js';
```

(continues on next page)

(continued from previous page)

```
import Module from 'opensphere/src/os/ui/module.js';
import SingleUrlProviderImportCtrl from 'opensphere/src/os/ui/singleurlproviderimport.
↪js';

import Tileserver from './tileserver.js';
import {ID} from './index.js';


/**
 * The Tileserver import directive
 * @return {angular.Directive}
 */
export const directive = () => ({
  restrict: 'E',
  replace: true,
  templateUrl: ROOT + 'views/forms/singleurl.html',
  controller: Controller,
  controllerAs: 'ctrl'
});

/**
 * The element tag for the directive.
 * @type {string}
 */
export const directiveTag = 'tileserver';

/**
 * Add the directive to the module
 */
Module.directive(directiveTag, [directive]);

/**
 * Controller for the Tileserver server import dialog
 * @unrestricted
 */
export class Controller extends SingleUrlProviderImportCtrl {
  /**
   * Constructor.
   * @param {!angular.Scope} $scope
   * @param {!angular.JQLite} $element
   * @ngInject
   */
  constructor($scope, $element) {
    super($scope, $element);

    const file = /** @type {os.file.File} */ ($scope['config']['file']);
    $scope['config']['url'] = file ? file.getUrl() : this.getUrl();
    $scope['urlExample'] = 'https://www.example.com/somepath/index.json';
    $scope['config']['type'] = ID;
    $scope['config']['label'] = this.getLabel() || 'Tileserver';
  }

  /**
   * @inheritDoc
   */
  getDataProvider() {
    const dp = new Tileserver();
```

(continues on next page)

(continued from previous page)

```
    dp.configure(this.scope['config']);
    return dp;
  }

  /**
   * @inheritDoc
   */
  getUrl() {
    if (this.dp) {
      const url = /** @type {Tileserver} */ (this.dp).getUrl();
      return url || '';
    }

    return '';
  }

  /**
   * @return {string}
   */
  getLabel() {
    if (this.dp) {
      const label = /** @type {Tileserver} */ (this.dp).getLabel();
      return label || '';
    }

    return '';
  }
}
```

Note that we do not have our own UI template. We are reusing one from OpenSphere and just overriding a couple of functions in the controller.

Now let's hook that up in our plugin.

Listing 43: `src/plugin/tileserver/tileserverplugin.js`

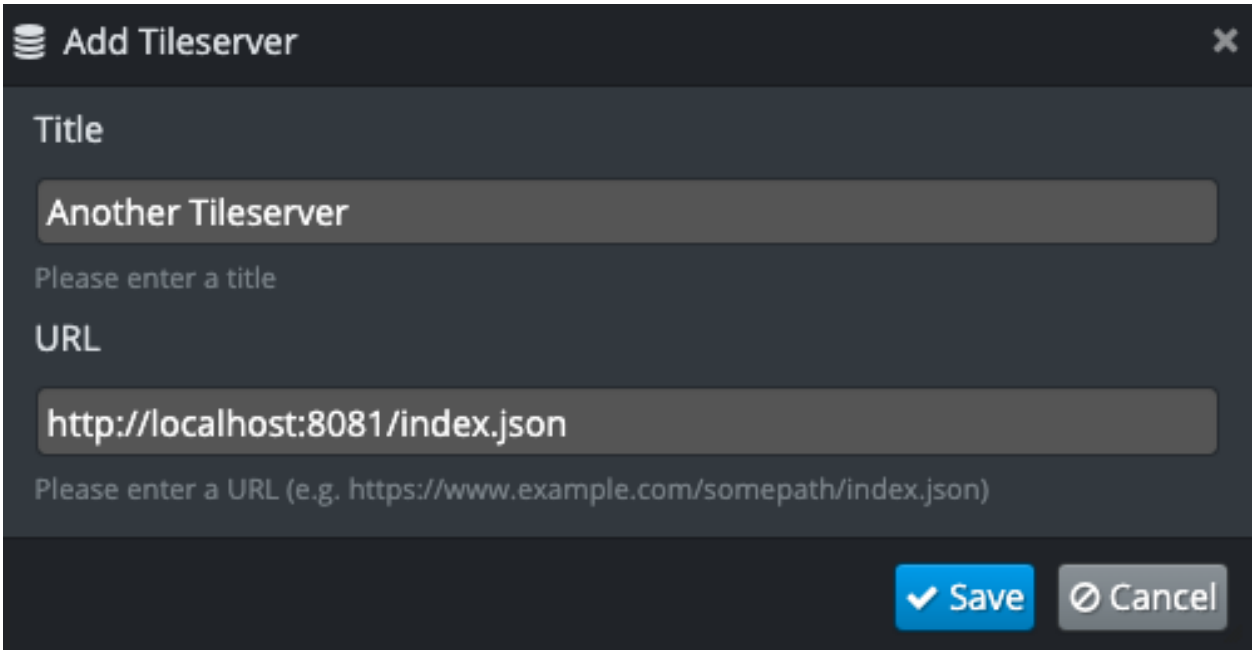
```
goog.declareModuleId('plugin.tileserver.TileserverPlugin');

import './mime.js';

import ConfigDescriptor from 'opensphere/src/os/data/configdescriptor.js';
import DataManager from 'opensphere/src/os/data/datamanager.js';
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import ImportManager from 'opensphere/src/os/ui/im/importmanager.js';
import ProviderImportUI from 'opensphere/src/os/ui/providerimportui.js';

import Tileserver from './tileserver.js';
import {directiveTag as importUi} from './tileserverimport.js';
import {ID} from './index.js';

/**
 * Provides Tileserver support
 */
export default class TileserverPlugin extends AbstractPlugin {
```

(continues on next page)

(continued from previous page)

```
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const dm = DataManager.getInstance();
    dm.registerProviderType(new ProviderEntry(
        ID, // the type
        Tileserver, // the class
        'Tileserver', // the title
        'Tileserver layers' // the description
    ));

    dm.registerDescriptorType(ID, ConfigDescriptor);

    const im = ImportManager.getInstance();
    // The argument to ProviderImportUI is the directive. A simple one like this will
    // be expanded under the hood to '<tileserver></tileserver>'. However, you can
    // define your own full markup and pass it in if you like.
    im.registerImportUI(ID, new ProviderImportUI(importUi));
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TileserverPlugin());
```

Save and run the build. Open the debug instance and go to Settings > Data Servers. You can now hit the little view icon on the Tileserver provider in that list. In addition, you can go to Add Data > Import File/URL and add the URL to the `index.json`. It will pop up a UI for you to edit the title and the URL, as shown below:



Saving this should save a new persistent copy of the provider in Data Servers. Edit and Delete should also work as expected. Note that the user cannot edit or delete providers that are configured in settings.

The final step is getting the provider type to appear in the Data Servers > Add Server UI. This UI enumerates supported server types to allow importing a specific type.

First we'll create a directive to display the import form without the additional window content.

Listing 44: `src/plugin/tileserver/tileserverimport.js`

```
goog.declareModuleId('plugin.tileserver.TileserverImportUI');

import {ROOT} from 'opensphere/src/os/os.js';
import Module from 'opensphere/src/os/ui/module.js';
import SingleUrlProviderImportCtrl from 'opensphere/src/os/ui/singleurlproviderimport.
↪js';

import Tileserver from './tileserver.js';
import {ID} from './index.js';


/**
 * The Tileserver import directive
 * @return {angular.Directive}
 */
export const directive = () => ({
  restrict: 'E',
  replace: true,
  templateUrl: ROOT + 'views/forms/singleurl.html',
  controller: Controller,
  controllerAs: 'ctrl'
});

/**
 * The element tag for the directive.
```

(continues on next page)

(continued from previous page)

```
 * @type {string}
 */
export const directiveTag = 'tileserver';

/**
 * Add the directive to the module
 */
Module.directive(directiveTag, [directive]);

/**
 * The Tileserver import directive
 * @return {angular.Directive}
 */
export const formDirective = () => {
  const directive = directive();
  directive.templateUrl = ROOT + 'views/forms/singleurlform.html';
  return directive;
};

/**
 * The element tag for the directive.
 * @type {string}
 */
export const formDirectiveTag = 'tileserverform';

/**
 * Add the directive to the module
 */
Module.directive(formDirectiveTag, [formDirective]);

/**
 * Controller for the Tileserver server import dialog
 * @unrestricted
 */
export class Controller extends SingleUrlProviderImportCtrl {
  /**
   * Constructor.
   * @param {!angular.Scope} $scope
   * @param {!angular.JQLite} $element
   * @ngInject
   */
  constructor($scope, $element) {
    super($scope, $element);

    const file = /** @type {os.file.File} */ ($scope['config']['file']);
    $scope['config']['url'] = file ? file.getUrl() : this.getUrl();
    $scope['urlExample'] = 'https://www.example.com/somepath/index.json';
    $scope['config']['type'] = ID;
    $scope['config']['label'] = this.getLabel() || 'Tileserver';
  }

  /**
   * @inheritDoc
   */
  getDataProvider() {
    const dp = new Tileserver();
    dp.configure(this.scope['config']);
```

(continues on next page)

(continued from previous page)

```
    return dp;
  }

  /**
   * @inheritDoc
   */
  getUrl() {
    if (this.dp) {
      const url = /** @type {Tileserver} */ (this.dp).getUrl();
      return url || '';
    }

    return '';
  }

  /**
   * @return {string}
   */
  getLabel() {
    if (this.dp) {
      const label = /** @type {Tileserver} */ (this.dp).getLabel();
      return label || '';
    }

    return '';
  }
}
```

Then we'll register the server type with the form UI.

Listing 45: `src/plugin/tileserver/tileserverplugin.js`

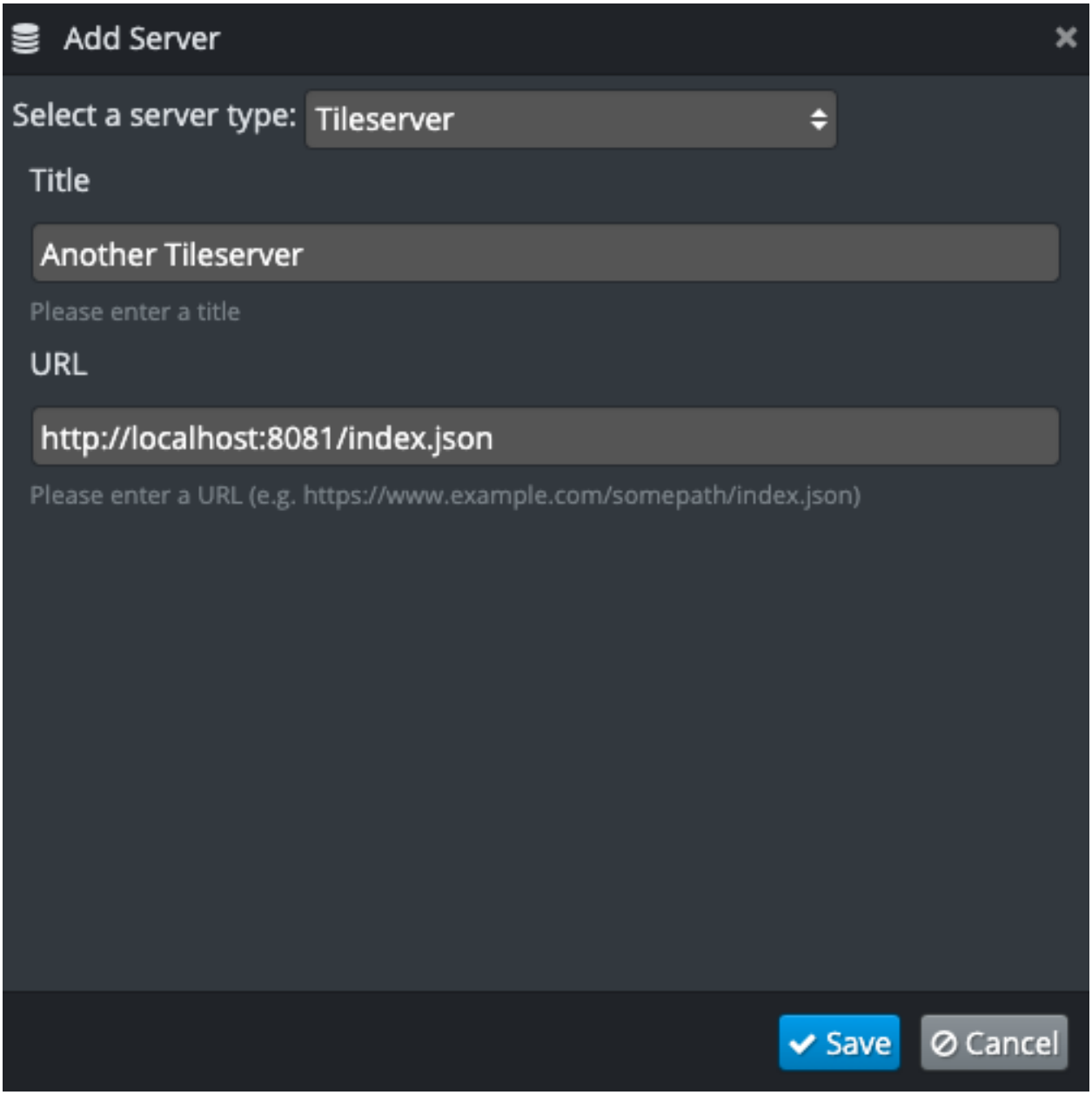
```
goog.declareModuleId('plugin.tileserver.TileserverPlugin');

import './mime.js';

import ConfigDescriptor from 'opensphere/src/os/data/configdescriptor.js';
import DataManager from 'opensphere/src/os/data/datamanager.js';
import ProviderEntry from 'opensphere/src/os/data/providerentry.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import ImportManager from 'opensphere/src/os/ui/im/importmanager.js';
import ProviderImportUI from 'opensphere/src/os/ui/providerimportui.js';

import Tileserver from './tileserver.js';
import {directiveTag as importUi, formDirectiveTag as formUi} from './
↪tileserverimport.js';
import {ID} from './index.js';

/**
 * Provides Tileserver support
 */
export default class TileserverPlugin extends AbstractPlugin {
  /**
   * Constructor.
```

(continues on next page)

(continued from previous page)

```
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const dm = DataManager.getInstance();
    dm.registerProviderType(new ProviderEntry(
        ID, // the type
        Tileserver, // the class
        'Tileserver', // the title
        'Tileserver layers' // the description
    ));

    dm.registerDescriptorType(ID, ConfigDescriptor);

    const im = ImportManager.getInstance();
    // The argument to ProviderImportUI is the directive. A simple one like this will
    // be expanded under the hood to '<tileserver></tileserver>'. However, you can
    // define your own full markup and pass it in if you like.
    im.registerImportUI(ID, new ProviderImportUI(importUi));

    // Register the server type with the import manager so it will appear in the Add
↪Server UI.
    im.registerServerType(ID, {
      type: ID,
      formUi,
      label: 'Tileserver'
    });
  }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TileserverPlugin());
```

Reload the debug application, then go to Settings > Data Servers > Add Server. You should now see a Tileserver option in the dropdown that will display the import UI.



That's pretty much it for providers. If you want to connect to a provider using formats not already supported by OpenSphere, then follow the *Parser* and *Layer Config* sections of the *File Type Plugin Guide*.

6.3 New Server/Provider Types

Data providers such as ArcGIS, Geoserver, and others are implemented as plugins. If you want to extend OpenSphere to support an additional server type or protocol (like OGC Catalog Service for the Web - CSW), this is the type of plugin you need to implement. Each of them adds:

- The provider (implementation of os.data.IDataProvider), which is responsible for querying the server and creating descriptors for the available layers
- Content Type and/or URL detection for adding new servers of that type, which in turn activates...

- A form for letting the user add a new server of that type
- A layer type if the server has a custom format (e.g. Arc REST service JSON). If the service returns already-implemented formats such as GeoJSON, this is not necessary.

6.4 Search Providers

Search providers (used via the search box in the top right) can search for places, layers, documents, or anything else. The Google Places and Descriptor Search plugins are good examples of search providers.

6.5 Plugin Examples

See opensphere-plugin-example for an example of setting up an external (or perhaps private) plugin. That is mostly boilerplate intended to get a project started and does not have much in the way of actual code.

Although OpenSphere supports external plugins, it has several "in-tree" plugins which provide functionality that we consider core to the application. These are all useful examples for plugin development, as well as their main purpose of providing useful functionality.

- area - adds the ability to import specific file types as areas instead of data layers
- audio - adds the ability to import custom audio files for different alert noises
- capture - adds the ability to record screenshots and GIF recordings of the timeline
- featureaction - adds the ability to run local filter functions on data import that takes the resulting items and applies an action (like changing the style)
- overview - adds the small overview map to the top right
- params - adds the ability to right-click layers and change the request parameters sent to the server
- places - integrates heavily with editable KML from the KML plugin to provide a "Saved Places" layer
- suncalc - adds the ability to show Sun/Moon info in a dialog (right-click on the map), in addition to the day/night lightstrip shown in the timeline
- vectortools - provides the copy/merge/join options for vector layers
- weather - simple plugin just adds an entry to the coordinate menu that launches a weather forecast for that location
- xyz - provides support for XYZ (zoom, X/Y) tile layers

Note that some of the plugins have corresponding views and tests.

CHAPTER 7

Settings Guide

OpenSphere's base settings live in `config/settings.json`. Unfortunately, JSON does not allow comments.

7.1 Basics

The entire settings tree falls under two categories:

```
{
  "admin": {
    ...
  },
  "user": {
    ...
  }
}
```

7.1.1 admin

Anything from `admin` is loaded "fresh" (but still follows browser caching directives) every time the application starts. The vast majority of all config goes here.

7.1.2 user

The `user` section is for specifying defaults for values that the user can otherwise permanently change in their settings. Developers can specify defaults in code:

```
// the second argument is essentially the application default
var bgColor = /** @type {string} */ (os.settings.get('bgColor', 'black'));
```

That value may never actually show up in `config/settings.json`, but sysadmins can change the default by placing a value like so:

```
{
  "user": {
    "bgColor": "white"
  }
}
```

Assuming the user has a GUI to change that value, they can still change it to whatever they like, but the default will now be `"white"`.

7.2 Merging

When OpenSphere is built, `settings.json` files from all over are merged into one final file which is placed in `opensphere/dist/opensphere/config/settings.json`. Take the following workspace example:

```
workspace/
  opensphere/
  opensphere-config-deployment-base/
  opensphere-config-deployment-specific/
  opensphere-plugin-x/
  opensphere-plugin-y/
```

By default, the merge order of the configs is defined in the order in which they are resolved. For the example above, this naturally results in the following merge order:

```
[
  // our base build project
  "opensphere",

  // plugins are resolved next
  "opensphere-plugin-x",
  "opensphere-plugin-y",

  // followed by config last
  "opensphere-config-deployment-base",
  "opensphere-config-deployment-specific"
]
```

You can see the merge order in `.build/settings-debug.json`, which is what the debug build output uses to load the files in the proper order.

If this order is not satisfactory, each project can define its own merge priority in `package.json:build.priority`.

7.3 Merge Values

The merge in the build is performed entirely by the config plugin of the resolver project.

Only objects accept merges. Everything else is a replacement:

```
var original = {
  "name": "Katie",
  "age": 29,
  "interests": ["dogs", "skiing"],
```

(continues on next page)

(continued from previous page)

```
  "likesColors": {
    "blue": true,
    "orange": false
  }
};

var newInfo = {
  "name": "Katie Smith",
  "age": 30,
  "interests": ["netflix"],
  "height": 150,
  "likesColors": {
    "purple": true,
    "orange": true
  }
};

// merge newInfo to original results in
var merged = {
  "name": "Katie Smith",
  "age": 30,
  "interests": ["netflix"],
  "height": 150,
  "likesColors": {
    "blue": true,
    "orange": true,
    "purple": true
  }
};
```

To delete a value, simply assign the value `"__delete__"`:

```
var moreInfo = {
  "likesColors": "__delete__"
}

// merge moreInfo to our previously merged object results in
var merged2 = {
  "name": "Katie Smith",
  "age": 30,
  "interests": ["netflix"],
  "height": 150
};
```

7.4 Settings

Here we will go through some of the most important settings individually. If you have any questions on a more minor one, let us know and we will try to get to it soon.

7.4.1 proxy

`admin.proxy`

The proxy is a failover for getting around mixed content and CORS warnings/errors from other servers.

- `url` The URL to the proxy; must contain `{url}` e.g. `https://cors-anywhere.herokuapp.com/{url}`
- `methods` The list of http methods supported by the proxy. e.g. `["GET", "POST", ...]`
- `schemes` The schemes supported by the proxy. e.g. `["http", "https"]`
- `encode` whether or not to URL-encode the entire URL when replacing `{url}`. Defaults to `true`.

Some items can be configured to use the proxy by default (such as basemaps and some providers). However, for most requests, they will first be tried as a normal request and only try the proxy after that request fails.

7.4.2 providers

`admin.providers`

The providers section is the meat of any OpenSphere configuration. This provides all of the data available to the user by default. While they can certainly add their own data servers, a well-curated list is much more likely to keep users coming back.

Usage:

```
{
  "admin": {
    "providers": {
      "unique-id-1": {
        "type": "geoserver", // or any provider type
        // ... rest of provider-specific config
      },
      // ... more providers
    }
  }
}
```

The design there should be simple and clear. However, let's do a specific example. The `config/settings.json` file in OpenSphere itself does a good job of showing how to set up the `basemap` provider. So we will add a couple of others:

```
{
  "admin": {
    "providers": {
      "arc-sample-server": {
        "type": "arc",
        "label": "ArcGIS Online",
        "url": "//services.arcgisonline.com/ArcGIS/rest/services/"
      },
      "demo-geoserver": {
        "type": "geoserver",
        "label": "Demo Geoserver",
        "url": "https://demo.geo-solutions.it/geoserver/ows"
      }
    }
  }
}
```

7.4.3 plugins

`admin.plugins`

This is an object map of plugin IDs to booleans that allows you to disable a plugin entirely in config rather than having to build a new version of the application without that plugin.

Say we wanted to disable KML for some reason:

```
{
  "admin": {
    "plugins": {
      "kml": false
    }
  }
}
```

7.4.4 baseProjection

`user.baseProjection`

The `baseProjection` sets the default projection of the application. This projection should have a corresponding set of default map layers configured in the `basemap` provider. OpenSphere ships with support for EPSG:4326 and EPSG:3857 out of the box. Other projections can be added via config.

Users can change this value in Settings > Map > Projection or by adding a tile layer that is in a projection other than the current projection (assuming that `enableReprojection` is `false`).

```
{
  "user": {
    "baseProjection": "EPSG:4326"
  }
}
```

7.4.5 metrics

`admin.metrics`

OpenSphere has a metrics API that can be used to gather stats about usage. We want to stress that these metrics are *not sent anywhere*. If you would like to have your metrics sent somewhere, you will have to write a plugin to upload them and add that to your OpenSphere build.

However, for the overly paranoid:

```
{
  "admin": {
    "metrics": {
      "enabled": false
    }
  }
}
```

7.4.6 terrain

`admin.providers.basemap.maps.terrain`

OpenSphere supports a number of terrain formats out of the box, and more can be added via plugins. By default, the following are supported:

- STK Terrain Server

- WMS Tiles (`image/bil` only)
- Cesium World Terrain (requires a Cesium Ion access token)

Enabling terrain in OpenSphere requires adding a basemap provider to settings with the requisite configuration for the terrain provider type. All providers should set `"type": "Terrain"` to identify it as a terrain server, and set `baseType` to one of `cesium`, `wms`, or `cesium-ion`. Examples of configuring each are as follows.

STK Terrain Server

```
{
  "admin": {
    "providers": {
      "basemap": {
        "maps": {
          "terrain": {
            "type": "Terrain",
            "baseType": "cesium",
            "options": {
              "url": "<server url>"
            },
          }
        }
      }
    }
  }
}
```

WMS Tiles

```
{
  "admin": {
    "providers": {
      "basemap": {
        "maps": {
          "terrain": {
            "type": "Terrain",
            "baseType": "wms",
            "options": {
              "url": "<server url>"
              "layers": [{
                "layerName": "<elevation layer>",
                "minLevel": 0,
                "maxLevel": 10
              }]
            }
          }
        }
      }
    }
  }
}
```

Where the *<server url>* would be something like *https://yourgeoserver:8080/geoserver/ows* and the *layers* entries have corresponding layer names and resolution ranges.

Cesium World Terrain

A Cesium Ion account is required to use Cesium World Terrain. After signing in to an Ion account:

1. Select a Cesium World Terrain asset from My Assets.
2. Set the `options.assetId` value to the asset ID (displayed in the code example).
3. Navigate to Access Tokens.
4. Set the `options.accessToken` value to a token with read access for the terrain asset.

Example:

```
{
  "admin": {
    "providers": {
      "basemap": {
        "maps": {
          "terrain": {
            "type": "Terrain",
            "baseType": "cesium-ion",
            "options": {
              "accessToken": "<ion access token>",
              "assetId": 1
            }
          }
        }
      }
    }
  }
}
```

7.4.7 columnActions

`admin.columnActions`

Column Actions are specific actions that are performed when displaying field data from a feature.

The only action Opensphere currently supports is the `UrlColumnAction`. When displaying feature information, this action will scan column names and values via regular expression to create a clickable link.

The value of the matched field will be substituted for `%s` in the `"action"` string.

Example:

```
{
  "admin": {
    "columnActions": {
      "googleMapsAction": {
        "type": "url",
        "description": "Create a link to a location on Google Maps.",
        "regex": {
          "col": "^DD_LAT_LON$",
          "val": "[\-\d]\d+\.\d+",
          "search": "\w",
          "replace": ","
        },
        "action": "https://www.google.com/maps/@%s"
```

(continues on next page)

(continued from previous page)

```
        }
      }
    }
}
```

One of "col" or "val" is required. "search" and "replace" can be used to transform the text before substituting into the URL, but are entirely optional.

While this is only executed on visible cells in Slickgrid, it is applied to every field's column and value and may cause performance problems depending on the complexity and specificity of the regular expression. If you know specific columns that you want to apply actions to, and know their format in advance, it may be better to simply match on the exact column name.

CHAPTER 8

Deployment Guide

8.1 Build

Our builds zip up the contents of the `dist` folder after `yarn run build` completes. This results in a zip directory structure like:

```
{appName}/
  config/
    settings.json
  v{buildtime}/
    ... code
  index.html
```

Generally, this build does not include any deployment-specific config. Since we have multiple deployments, and we do not wish to run the (much longer) full build for each separate deployment, we have other build jobs which grab the `settings.json` file out of this zip, merge in the deployment-specific config, and save it as `{app}-{version}-{deployment}.zip`. However, if you only have a single deployment, you may wish to include your deployment config as a config project during the main build.

8.2 Settings

See our *Settings Guide* to get familiar with the main settings to configure for your deployment, such as remote servers and base map layers.

8.3 HTTP Server

We highly recommend the use of nginx to serve the application. However, since the application is entirely static html, js, css, etc. it can be served with any HTTP server. The following config examples will be using nginx.

8.3.1 Define the WASM type

Our build includes some `*.wasm` files. Browsers complain in the console if the mime type is not set properly for these.

```
http {
  ...

  include mime.types;
  types {
    application/wasm wasm;
  }
}
```

8.3.2 Enable GZIP

This should go without saying. Nevertheless:

```
 http {
   ...

   gzip on;
   gzip_disable "msie6";
   gzip_min_length 1100;
   gzip_vary on;
   gzip_buffers 16 8k;
   gzip static always;
   gzip_types text/plain text/css text/js text/xml text/html text/javascript␣
↪application/javascript application/x-javascript application/json application/xml␣
↪application/xml+rss application/wasm;
}
```

8.3.3 Caching

We should allow the browser to cache config for a couple of hours and the version folder (`v{buildtime}`) for as long as possible.

```
http {
  ...
  server {
    ...
    # allow clients to cache config for a couple of hours
    location ^~ /opensphere/config/ {
      add_header 'Cache-Control' 'max-age=7200, public' always;
    }

    # anything under the version directories should be cached for the maximum (one␣
↪year)
    location ^~ /opensphere/v\d+/ {
      add_header 'Cache-Control' 'max-age=31557600, public' always;
    }
  }
}
```

Note: If you dropped the app name (such that the application is available directly on `yourdomain.com/` rather than `yourdomain.com/opensphere`), you will need to drop the `/opensphere` off the location directives in the above example.

8.3.4 Deploying

The directory structure is designed to simply be unzipped over old installs. This allows users with cached `index.html` pages to previous versions to work until they request `index.html` again after that cache expires (generally 20-30 minutes). Therefore, just `cd` to the document root and unzip the file (and ensure that permissions are set correctly for your server, of course).

8.4 API Keys

Some data or search providers require an API Key to function. For development, you can merely drop the API Key directly in config:

Warning: The following is acceptable for dev config but should not be done in production! Anyone who uses the application will have access to the API Key.

```
{
  "admin": {
    "plugin": {
      "myplugin": {
        "url": "https://api.somecoolsite.com/search?q={q}&api_key=SOMETHING"
      }
    }
  }
}
```

For production, it is better to hide that API behind a proxy that you control.

```
http {
  ...
  server {
    ...
    location /somecoolsite/search {
      set $args $args&api_key=SOMETHING;
      proxy_pass https://api.somecoolsite.com/search;
    }
  }
}
```

And the production config:

```
{
  "admin": {
    "plugin": {
      "myplugin": {
        "url": "/somecoolsite/search?q={q}"
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

CHAPTER 9

Developers' Cookbook

The Developers' Cookbook provides snippets and small examples of how to use parts of the OpenSphere API.

For extended examples, see the Guides.

9.1 Add Layer Page

9.1.1 Problem

We want to support links that add layers or files to OpenSphere.

9.1.2 Solution

The OpenSphere `addlayer.html` page. This page is built alongside OpenSphere and takes a URI-encoded JSON string in the fragment, decodes it and passes it over to OpenSphere (or opens OpenSphere if it is not already up).

9.1.3 Discussion

File URL

Use a path of:

```
'example.com/path/to/opensphere/addlayer.html#file=' +
    encodeURIComponent(fileUrl);
```

This prompts the user to import the file in the application. However, for many files, you may want to set all of your options up ahead of time so that the user does not have to figure them out in the Import Wizard.

Therefore, you may want to use Layer Configs instead.

Layer Config

Use a path of:

```
'example.com/path/to/opensphere/addlayer.html#' +
    encodeURIComponent(JSON.stringify(conf));
```

where `conf` is a JSON layer config. This is not the proper place to go into all the available layer configs, but one of the simplest possible configs is:

```
{
  "type": "WMS",
  "url": "https://demo.geo-solutions.it/geoserver/ows",
  "params": "LAYERS=topp:states",
  "projection": "EPSG:4326",
  "title": "US States"
}
```

However, you may want to add other common options such as:

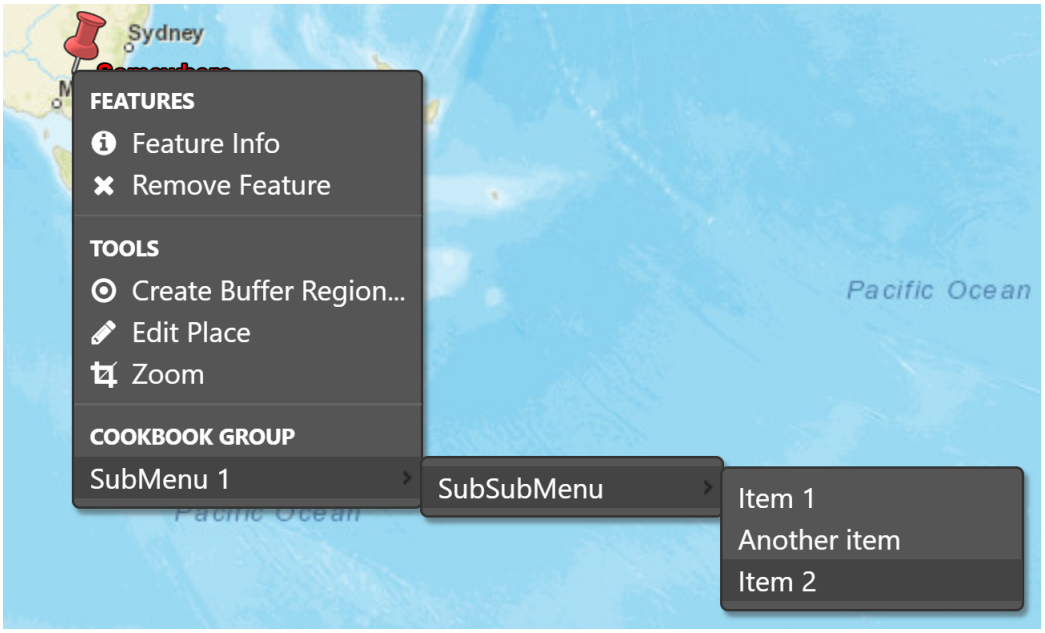
```
{
  "type": "WMS",
  "url": "https://demo.geo-solutions.it/geoserver/ows",
  "params": "LAYERS=topp:states",
  "title": "US States",
  "projection": "EPSG:4326",

  // give them some credit
  "description": "States colored by population provided by geo-solutions.it",
  "attributions": ["geo-solutions.it"],
  "provider": "geo-solutions.it demo GeoServer",

  // don't load tiles outside of this extent
  "extent": [-124.731422, 24.955967, -66.969849, 49.371735],
  "extentProjection": "EPSG:4326",

  // whether or not use the configured proxy (if any)
  "proxy": false,

  // CORS is annoying; We try to auto-detect the proper value, but
  // it can also be set explicitly
  "crossOrigin": "anonymous",

  // you can also set default style parameters like opacity, color, etc.
  "opacity": 0.5
}
```

Every layer that can be shown by OpenSphere has a layer config that can be used here. Note, however, that the layer is transient and will not be saved between sessions (as of now, anyway). You can save a state file in OpenSphere to preserve them if you like.

9.2 SubMenu

9.2.1 Problem

Your plugin needs to show additional menu entries in a context menu (e.g. the Spatial menu that appears when you right click on a feature), but you don't want to modify OpenSphere menu code for your specific case.



9.2.2 Solution

Extend the existing menu with Group, SubMenu and Item elements to provide your plugin specific functionality.

Listing 1: SubMenu Cookbook example - menu creation

```
const menu = spatial.getMenu();
if (menu) {
  const root = menu.getRoot();
  let group = root.find(MYGROUP);
  if (!group) {
    group = root.addChild({
      type: MenuItemType.GROUP,
      label: MYGROUP,
      tooltip: 'Added by cookbook submenu example',
      beforeRender: shouldShowGroup
    });
  }
  const submenu1 = group.addChild({
    type: MenuItemType.SUBMENU,
    label: 'SubMenu 1'
  });
  // Submenus can be nested
  const submenu2 = submenu1.addChild({
    type: MenuItemType.SUBMENU,
    label: 'SubSubMenu',
```

(continues on next page)

(continued from previous page)

```
        // You can specify child menu items directly
        children: [{
          type: MenuItemType.ITEM,
          label: 'Item 1',
          sort: 10,
          handler: handleItem1
        }, {
          type: MenuItemType.ITEM,
          eventType: EventType.DO_SOMETHING,
          label: 'Item 2',
          sort: 30,
          handler: handleItem,
          beforeRender: visibleIfIsPointInSouthernHemisphere
        }]
      });

      // You can also add items programmatically
      submenu2.addChild({
        type: MenuItemType.ITEM,
        eventType: EventType.DO_ANOTHER_THING,
        label: 'Another item',
        sort: 20,
        handler: handleItem
      });
    }
```

9.2.3 Discussion

The Spatial menu (`os.ui.menu.spatial.MENU`) is extensible from plugins. You can then attach groups, separators, submenus and items (plain items, checkboxes, or radio buttons) to the root, or to sub-items. In the image and code, a group item is added to the root, then submenus are nested below that group, and some items are available for selection.

If your plugin has all items known in advance, its possible to use the `children` property to nest the whole structure, as shown here:

```
const menu = spatial.getMenu();
if (menu) {
  const root = menu.getRoot();
  let group = root.find(MYGROUP);
  if (!group) {
    group = root.addChild({
      type: MenuItemType.GROUP,
      label: MYGROUP,
      tooltip: 'Added by cookbook submenu example',
      beforeRender: shouldShowGroup,
      children:[{
        type: MenuItemType.SUBMENU,
        label: 'SubMenu 1',
        children:[{
          type: MenuItemType.SUBMENU,
          label: 'SubSubMenu',
          children: [{
            type: MenuItemType.ITEM,
```

(continues on next page)

(continued from previous page)

```
            label: 'Item 1',
            sort: 10,
            handler: handleItem1
          }, {
            type: MenuItemType.ITEM,
            eventType: EventType.DO_SOMETHING,
            label: 'Item 2',
            sort: 30,
            handler: handleItem,
            beforeRender: visibleIfIsPointInSouthernHemisphere
          }, {
            type: MenuItemType.ITEM,
            eventType: EventType.DO_ANOTHER_THING,
            label: 'Another item',
            sort: 20,
            handler: handleItem
          }]
        }]
      }]
    });
  }
```

This is equivalent - you can mix and match the `addChild` function calls and nested `children` array as needed.

Note that items will be enabled and visible by default, which might not make sense for your use. Instead, you may want to selectively enable, or not display, some menu items, depending on the menu context. The usual way to do this is to set a `beforeRender`, as in `visibleIfIsPointInSouthernHemisphere` which is implemented as:

Listing 2: SubMenu Cookbook example - selectable visibility

```
/**
 * If our item should be shown.
 * @param {Object|undefined} context The menu context.
 * @this {os.ui.menu.MenuItem}
 */
const visibleIfIsPointInSouthernHemisphere = function(context) {
  this.visible = ifIsPointInSouthernHemisphere(context);
};

/**
 * If feature associated with menu entry is a point in southern hemisphere.
 * @param {Object|undefined} context The menu context.
 * @return {boolean}
 */
const ifIsPointInSouthernHemisphere = function(context) {
  // Get feature associated with menu context
  const features = spatial.getFeaturesFromContext(context);
  if (features.length === 1) {
    const feature = features[0];
    const geom = feature.getGeometry();
    if (geom instanceof Point) {
      const coords = toLonLat(geom.getFlatCoordinates(), PROJECTION);
      if (coords[1] < 0) {
        return true;
      }
    }
  }
```

(continues on next page)

(continued from previous page)

```
  return false;
};
```

As the name suggests, one menu entry will only be shown if the feature geometry is a point located in the southern hemisphere (i.e. negative latitude).

9.2.4 Full code

Listing 3: SubMenu Cookbook example - Full code

```
goog.declareModuleId('plugin.cookbook_submenu.CookbookSubmenu');

import {PROJECTION} from 'opensphere/src/os/map/map.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import MenuItemType from 'opensphere/src/os/ui/menu/menuitemtype.js';
import * as spatial from 'opensphere/src/os/ui/menu/spatial.js';

const Point = goog.require('ol.geom.Point');
const {toLonLat} = goog.require('ol.proj');

const {default: MenuEvent} = goog.requireType('os.ui.menu.MenuEvent');


/**
 * Cookbook example of a submenu
 */
export default class CookbookSubmenu extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const menu = spatial.getMenu();
    if (menu) {
      const root = menu.getRoot();
      let group = root.find(MYGROUP);
      if (!group) {
        group = root.addChild({
          type: MenuItemType.GROUP,
          label: MYGROUP,
          tooltip: 'Added by cookbook submenu example',
          beforeRender: shouldShowGroup
        });
      }
      const submenu1 = group.addChild({
        type: MenuItemType.SUBMENU,
```

(continues on next page)

(continued from previous page)

```
        label: 'SubMenu 1'
      });
      // Submenus can be nested
      const submenu2 = submenu1.addChild({
        type: MenuItemType.SUBMENU,
        label: 'SubSubMenu',

        // You can specify child menu items directly
        children: [{
          type: MenuItemType.ITEM,
          label: 'Item 1',
          sort: 10,
          handler: handleItem1
        }, {
          type: MenuItemType.ITEM,
          eventType: EventType.DO_SOMETHING,
          label: 'Item 2',
          sort: 30,
          handler: handleItem,
          beforeRender: visibleIfIsPointInSouthernHemisphere
        }]
      });

      // You can also add items programmatically
      submenu2.addChild({
        type: MenuItemType.ITEM,
        eventType: EventType.DO_ANOTHER_THING,
        label: 'Another item',
        sort: 20,
        handler: handleItem
      });
    }
  }
}

/**
 * @type {string}
 */
const ID = 'cookbook_submenu';

/**
 * @type {string}
 */
const MYGROUP = 'Cookbook Group';

/**
 * Our event types
 * @enum {string}
 */
const EventType = {
  DO_SOMETHING: 'cookbook:do_y',
  DO_ANOTHER_THING: 'cookbook:do_x'
};

/**
 * If our example group should be shown.
 * @param {Object|undefined} context The menu context.
```

(continues on next page)

(continued from previous page)

```
 * @this {os.ui.menu.MenuItem}
 */
const shouldShowGroup = function(context) {
  // This shows always, and could just be omitted from the menu item definition.
  // Your code could use the menu context if needed.
  this.visible = true;
};

/**
 * If our item should be shown.
 * @param {Object|undefined} context The menu context.
 * @this {os.ui.menu.MenuItem}
 */
const visibleIfIsPointInSouthernHemisphere = function(context) {
  this.visible = ifIsPointInSouthernHemisphere(context);
};

/**
 * If feature associated with menu entry is a point in southern hemisphere.
 * @param {Object|undefined} context The menu context.
 * @return {boolean}
 */
const ifIsPointInSouthernHemisphere = function(context) {
  // Get feature associated with menu context
  const features = spatial.getFeaturesFromContext(context);
  if (features.length === 1) {
    const feature = features[0];
    const geom = feature.getGeometry();
    if (geom instanceof Point) {
      const coords = toLonLat(geom.getFlatCoordinates(), PROJECTION);
      if (coords[1] < 0) {
        return true;
      }
    }
  }
  return false;
};

/**
 * Process a menu item
 * @param {MenuEvent} event The menu event.
 */
const handleItem1 = function(event) {
  alert('cookbook_submenu item1 selected');
};

/**
 * Process a menu item
 * @param {MenuEvent} event The menu event.
 */
const handleItem = function(event) {
  // event provides context for the elected item
  const eventType = event.type;
  alert('cookbook_submenu item selected:' + eventType);
};

// add the plugin to the application
```

(continues on next page)

(continued from previous page)

```
160 PluginManager.getInstance().addPlugin(new CookbookSubmenu());
```

9.3 Logging

9.3.1 Problem

Your plugin needs to log different types of information to support debugging or usage metrics.

9.3.2 Solution

Use the OpenSphere logging framework. There are three parts to enable this - adding the logger, using the logger, and adding the applicable `goog.require` entries.

Listing 4: Logging Cookbook example - requires

```
1 const log = goog.require('goog.log');
2 const Level = goog.require('goog.log.Level');
3 const Logger = goog.requireType('goog.log.Logger');
```

Listing 5: Logging Cookbook example - adding the logger

```
1 /**
2  * Logger.
3  * @type {Logger}
4  */
5 const logger = log.getLogger('plugin.cookbook_logging.CookbookLogging');
```

Listing 6: Logging Cookbook example - using the logger

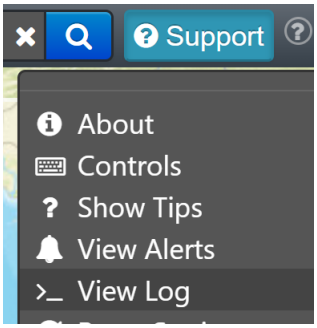
```
1   init() {
2     log.info(logger, 'Visible at INFO or below');
3     log.warning(logger, 'Visible at WARN or below');
4     log.log(logger, Level.FINEST, 'Visible only at FINEST or ALL');
5     log.error(logger, 'ALWAYS VISIBLE!');
6   }
```

9.3.3 Discussion

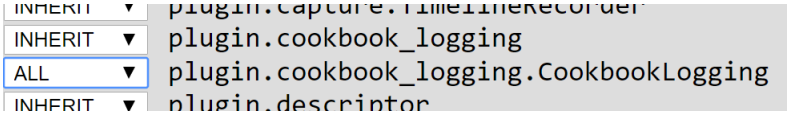
The OpenSphere logging framework is mostly based on the Closure logging library functions. The code above shows the two required argument form (which allows logging at error, warning, info and fine levels) as well as the three required argument `goog.log.log` form (which allows specifying of the log level for more options). Historically OpenSphere has only used the two required argument form (roughly half of the logging using `goog.log.error`, with `goog.log.warning`, `goog.log.info` and `goog.log.fine` sharing the other half reasonably evenly).

Tip: If you do need the `goog.log.log` form, use `goog.debug.Logger.Level` instead of `goog.log.Level` to specify the level, in order to avoid logging that works in a debug environment and throws exceptions in a production (minified / compiled) environment.

Having adding logging support to your plugin, you can access logs from within OpenSphere from the Support menu, using the View Logs entry:



Each logger that has been added will appear in the Options menu. The entries for our logger appear as:



Note that there are entries for `plugin.cookbook_logging` and `plugin.cookbook_logging.CookbookLogging`. This makes it easy to configure the appropriate logging levels for your plugin, and for lower level namespaces as needed.

9.3.4 Full code

Listing 7: Logging Cookbook example - Full code

```
goog.declareModuleId('plugin.cookbook_logging.CookbookLogging');

import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';

const log = goog.require('goog.log');
const Level = goog.require('goog.log.Level');
const Logger = goog.requireType('goog.log.Logger');


/**
 * Cookbook example of logging.
 */
export default class CookbookLogging extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    log.info(logger, 'Visible at INFO or below');
```

(continues on next page)

(continued from previous page)

```
    log.warning(logger, 'Visible at WARN or below');
    log.log(logger, Level.FINEST, 'Visible only at FINEST or ALL');
    log.error(logger, 'ALWAYS VISIBLE!');
  }
}

/**
 * Logger.
 * @type {Logger}
 */
const logger = log.getLogger('plugin.cookbook_logging.CookbookLogging');

/**
 * @type {string}
 */
export const ID = 'cookbook_logging';

// add the plugin to the application
PluginManager.getInstance().addPlugin(new CookbookLogging());
```

9.4 OpenStreetMap

9.4.1 Problem

You'd like to use an alternative base map, such as OpenStreetMap (OSM), for cartographic styling reasons or to provide a suitable base map on an isolated network.

9.4.2 Solution

Modify the config settings to specify an additional or alternative base map provider.

The following code shows how to change the config settings to add an additional basemap. Note that the existing "streetmap" and "worldimagery" parts, and other elements, are skipped to simplify the presentation. You can make other changes to meet your requirements, but removing existing items is not necessary to add a new provider.

Listing 8: `config/settings.json`

```
{
  "providers": {
    "basemap": {
      "maps": {
        "osm": {
          "title": "OpenStreetMap",
          "type": "BaseMap",
          "baseType": "XYZ",
          "provider": "OSM",
          "url": "//c.tile.openstreetmap.org/{z}/{x}/{y}.png",
          "minZoom": 2,
          "maxZoom": 19,
          "projection": "EPSG:3857",
          "tileSize": 256,
          "description": "OpenStreetMap rendered tiles.",
```

(continues on next page)

(continued from previous page)

```
          "attributions": ["© <a href=\"https://www.openstreetmap.org/copyright\">
↪OpenStreetMap</a> contributors"]
        }
      }
    }
  }
}
```

If you want the "osm" basemap to be shown by default, add it to the `defaults` section as shown here:

```
{
  "providers": {
    "basemap": {
      "type": "basemap",
      "defaults": {
        "EPSG:4326": [],
        "EPSG:3857": ["worldimagery", "streetmap", "osm"]
      }
    }
  }
}
```

9.4.3 Discussion

The XYZ plugin supports a standard, efficient way of transferring maps as gridded tiles at different zoom levels - sometimes called "Tile Map Service" or TMS. Note that this is not the same as Web Map Tile Service (WMTS).

Note: Review the terms of service for any tile server you are using, especially acceptable usage limits and attribution requirements.

The key parts of the configuration are the `url`, which includes placeholders for the zoom level, and X/Y indices for tiles. There are other tile servers available, which will have different URL content, potentially including requiring the placeholders in a different order. Note that the URL text does not include the protocol part. Also note that the URL is sometimes shown with `${z}` style placeholders, which are not supported by OpenSphere. However `%z` style placeholders are supported as an alternative to the `{z}` style.

If your provider supports multiple URLs (which is the case for most OSM styles), you can replace the `url` with `urls`, as shown below:

Listing 9: `config/settings.json`

```
{
  "providers": {
    "basemap": {
      "maps": {
        "osm": {
          "title": "OpenStreetMap",
          "type": "BaseMap",
          "baseType": "XYZ",
          "provider": "OSM",
          "urls": ["//a.tile.openstreetmap.org/{z}/{x}/{y}.png",
            "//b.tile.openstreetmap.org/{z}/{x}/{y}.png",
            "//c.tile.openstreetmap.org/{z}/{x}/{y}.png"],
```

(continues on next page)

(continued from previous page)

```
            "minZoom": 2,
            "maxZoom": 19,
            "projection": "EPSG:3857",
            "tileSize": 256,
            "description": "OpenStreetMap rendered tiles.",
            "attributions": ["© <a href=\"https://www.openstreetmap.org/copyright\">
↪OpenStreetMap</a> contributors"]
        }
      }
    }
  }
}
```

Another way to express that is `//{a-c}.tiles.example.com/osm/{z}/{x}/{y}.png`, where the `{a-c}` part will be expanded. You can use single letter (upper or lower case) or single number ranges (e.g. `{0-4}`) as appropriate to your server naming.

Other important values are the `minZoom` and `maxZoom` values, which specify the zoom levels that OpenSphere will show this base map at. Different OSM tile servers will support different zoom levels (e.g. the Humanitarian style is provided to zoom level 20).

You can also set up your own tile server, using the same system that OSM uses - see switch2osm or a more general server like GeoServer .

9.5 Audio Notification

9.5.1 Problem

Your plugin needs to provide some form of audio notification or audible alert.

9.5.2 Solution

Use the OpenSphere AudioManager. This is a singleton, and the *play()* function takes a label for the sound file to play:

Listing 10: AudioManager usage

```
1    const audioManager = AudioManager.getInstance();
2    audioManager.play('sound1');
```

There are three ways to provide the sound files - config settings, programmatically, or by user upload.

A config setting entry is shown below:

Listing 11: AudioManager sound file configuration

```
1  {
2    "admin": {
3      "sounds": { "sound1" : "sounds/cowbell.wav" }
4    }
5  }
```

A programmatic approach is shown below:

```
import {ROOT} from 'opensphere/src/os/os.js';
import AudioManager from 'opensphere/src/os/audio/audiomanager.js';

const audioManager = AudioManager.getInstance();
audioManager.addSound(ROOT + 'sounds/cowbell.wav', 'label');
audioManager.play('label');
```

User upload uses the normal Import Data dialog. Note that it only works in a standard browser environment if the target is a HTTP or HTTPS URL (different rules apply in a "wrapped" environment like [Electron](https://electronjs.org/)).

9.5.3 Discussion

The level of audio support, including the file formats and associated codecs that are supported, depends on browser capabilities.

In addition to the API shown earlier, AudioManager also supports muting the notifications. If your plugin makes use of audio notifications, we strongly suggest supporting global muting, as well as selective enable / disable of alerts.

```
audioManager.setMute(true);
const mute = audioManager.getMute(); // true - muted
audioManager.setMute(false); // now unmuted
```

9.5.4 Full code

Listing 12: Audio Notification Cookbook example - Full code

```
goog.declareModuleId('plugin.audioalert.AudioAlertPlugin');

import AudioManager from 'opensphere/src/os/audio/audiomanager.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';


/**
 * Cookbook example for playing an audio alert.
 */
export default class AudioAlertPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const audioManager = AudioManager.getInstance();
    audioManager.play('sound1');
  }
}
```

(continues on next page)

(continued from previous page)

```
29
30 /**
31  * @type {string}
32  */
33 const ID = 'audioalert';
34
35 // add the plugin to the application
36 PluginManager.getInstance().addPlugin(new AudioAlertPlugin());
```

9.6 Pelias Search (Forward geocoding)

9.6.1 Problem

You'd like to use the Pelias geocoder (or an equivalent online service, like geocode.earth).

This was previously also known as Mapzen Search, although those services are end-of-life.

9.6.2 Solution

Add suitable config settings for the service you want to enable.

The base application includes the Pelias plugin, but it is disabled until configured. The following code shows how to change the config settings for Pelias from geocode.earth, where your API key goes in place of the REPLACE_THIS part of the `url`:

Listing 13: `config/settings.json`

```
{
  "admin": {
    "plugin": {
      "pelias": {
        "geocoder": {
          "url": "https://api.geocode.earth/v1/search?api_key=REPLACE_THIS&text={s}",
          "extentParams": true,
          "extentThreshold": 300000,
          "focusPoint": true,
          "focusPointMinZoom": 5.0
        }
      }
    }
  }
}
```

Note: The rest of the `admin` settings, and the `user` settings, are not shown to make it easier to see. If you're using a standard config file, the `plugin` part is at the same level (peer item) to the `about` settings.

9.6.3 Discussion

If you have a local service, you do not need to provide an API key. For example, if your instance is running on localhost:3100 (standard for Pelias development installs), your `url` would likely be `http://localhost:3100/`

`v1/search?text={s}`.

The `extentParams` and `extentThreshold` parts are optional, and change the search URL sent to the Pelias service. `extentParams` enables (set false or leave out to disable) a rectangular boundary constraint on the search that corresponds to the map window, which is applied when the map extent is smaller (in metres) than the `extentThreshold` (or 200km if not specified). The user effect is that searches only include results that would appear on the map.

Similarly, the `focusPoint` and `focusPointMinZoom` parts are optional. `focusPoint` enables (set false or leave out to disable) a hint towards results that are close to the centre of the map. It only takes effect if the map is zoomed in to at least the `focusPointMinZoom` value (or 4.0 if not specified). Unlike the extent values that make a hard constraint, this is a hint on prioritisation, and results from long distances away may also be provided.

If you do need to use an API key, but don't want to expose it to every user, a reverse proxy server to add the API key to the query may be useful. One option is to the Apache mod_proxy and mod_rewrite capabilities. A typical configuration might look like:

```
SSLProxyEngine On
SSLProxyCheckPeerCN on
SSLProxyCheckPeerExpire on

RewriteEngine  on
RewriteRule ^/v1/search https://api.geocode.earth/v1/search?api_key=REPLACE_THIS [QSA,
↪P]
ProxyPassReverse "/v1/search" "https://api.geocode.earth/v1/search"
```

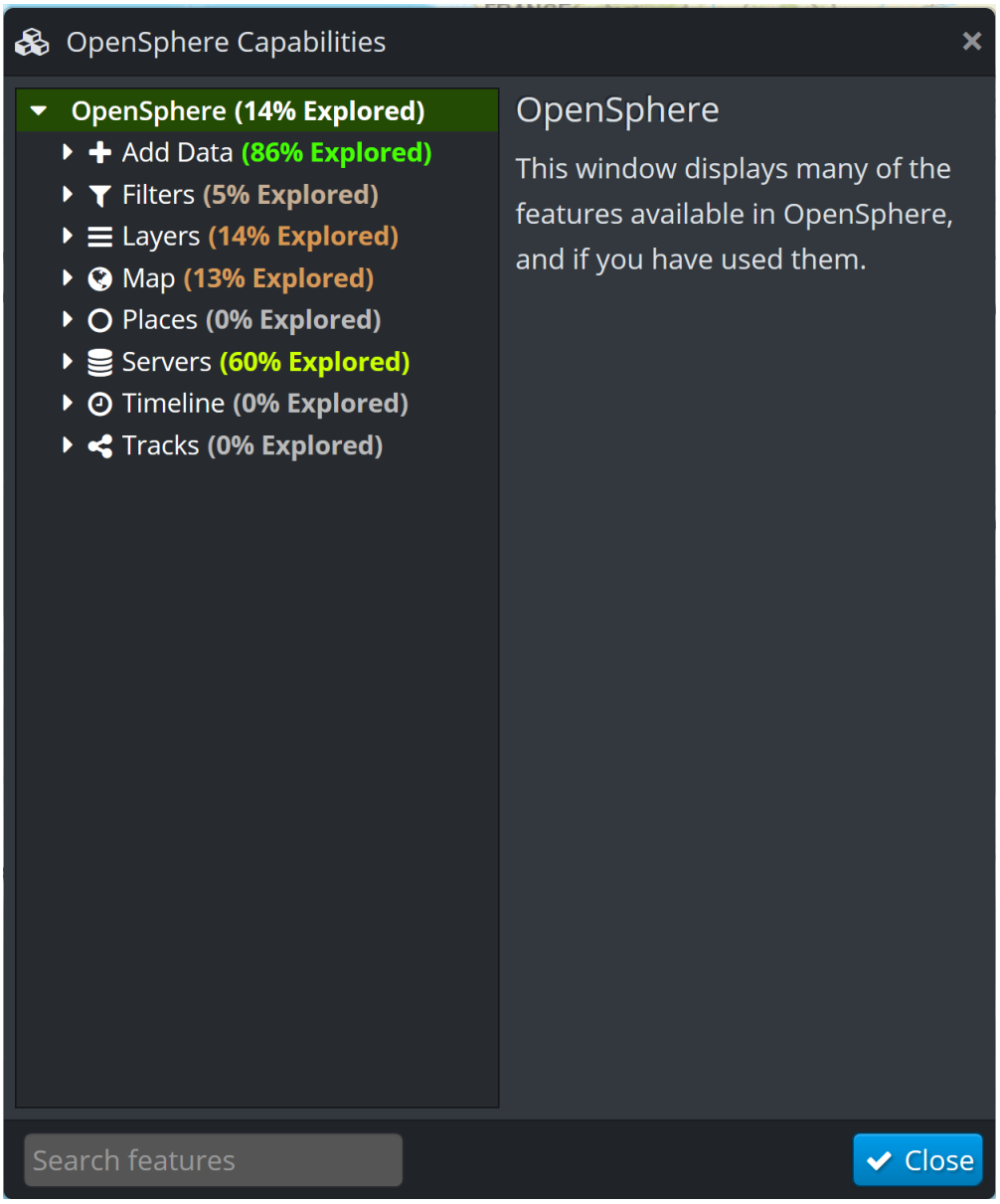
Note your API key goes in place of the REPLACE_THIS part of the `RewriteRule`. This will also need some modules to be loaded (if not already in place): proxy, proxy_http, rewrite and ssl. You can now use a `url` of something like `https://MY_SERVER/v1/search?text={s}` (or `http://MY_SERVER/v1/search?text={s}` if your server is not configured for HTTPS, which you probably should consider fixing) where the MY_SERVER part is replaced with your reverse proxy server hostname or IP address.

You should also consider the extent to which you need to secure your reverse proxy server. Consult the applicable server documentation for authentication options for your deployment scenario.

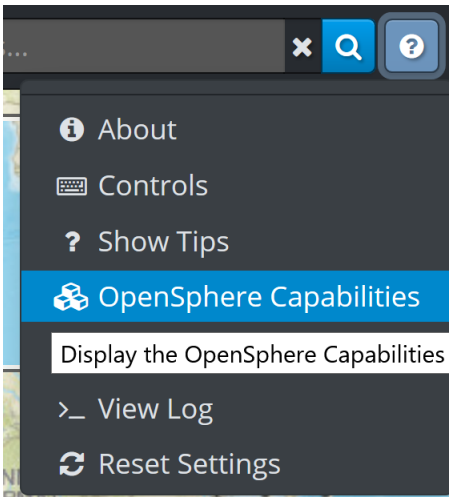
9.7 Metrics

9.7.1 Problem

You want your plugin to show up in the "Capabilities" dialog to help users discover your plugin features.



Note: This is available from the Support menu, in the top right corner of the main window.



9.7.2 Solution

Use the metrics API to provide your plugin specific functionality.

Listing 14: Metrics Cookbook example - menu creation

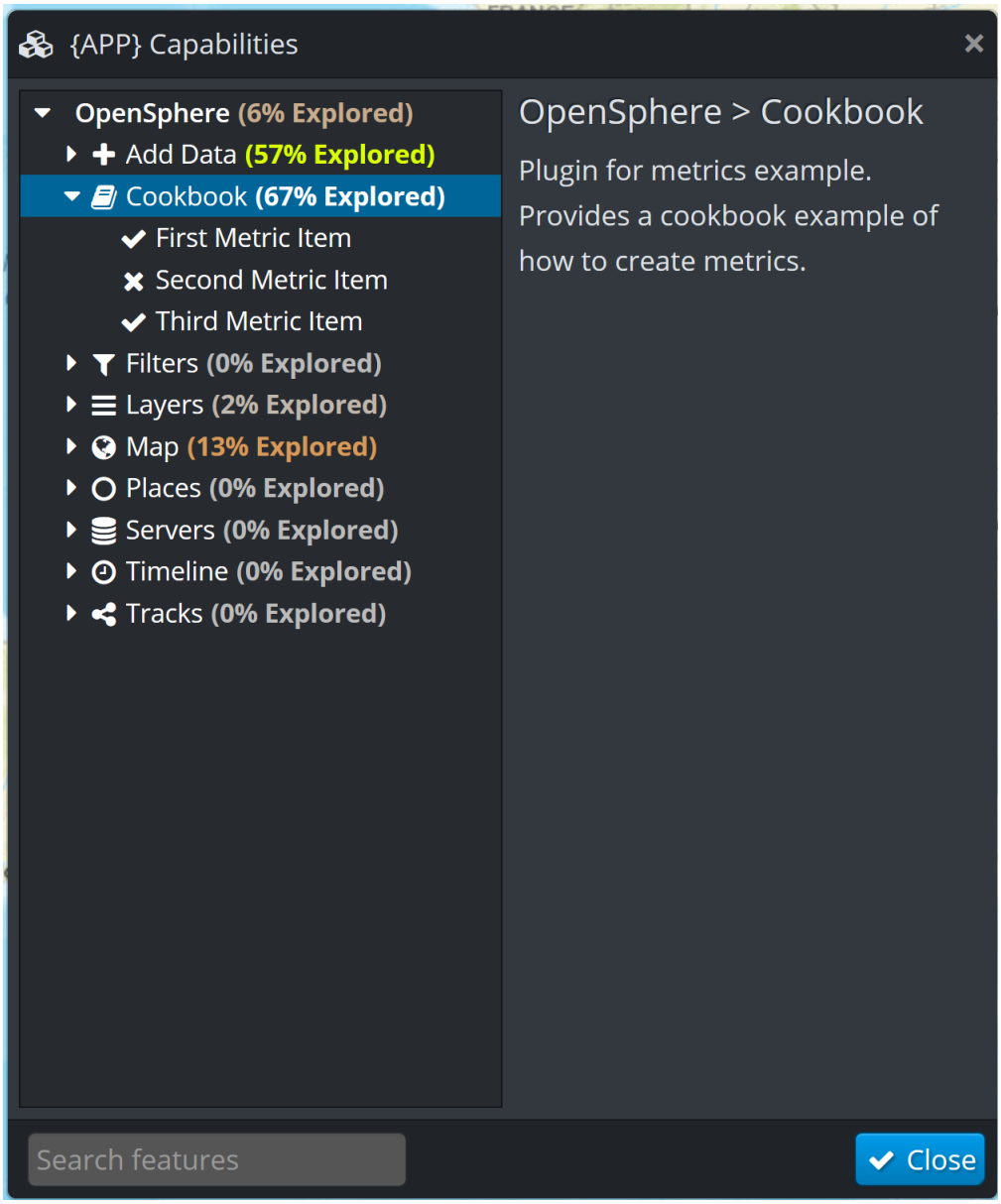
```
goog.declareModuleId('plugin.cookbook_metrics.CookbookMetrics');

import MetricsPlugin from 'opensphere/src/os/ui/metrics/metricsplugin.js';
import {Metrics} from './index.js';

/**
 */
export default class CookbookMetrics extends MetricsPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.setLabel('Cookbook');
    this.setIcon('fa fa-book');
    this.setCollapsed(true);
    this.setDescription('Plugin for metrics example.');

    const leaf = this.getLeafNode();
    this.addChild(leaf, {
      label: 'First Metric Item',
      description: 'This is an item.',
      key: Metrics.FIRST_THING
    });

    this.addChild(leaf, {
      label: 'Second Metric Item',
      description: 'Does something.',
      key: Metrics.SECOND_THING
    });

    this.addChild(leaf, {
```

(continues on next page)

(continued from previous page)

```
      label: 'Third Metric Item',
      description: 'Combination, programmatic.',
      key: Metrics.EXTRA_THING
    });
  }
}
```

The dialog will then have additional items



9.7.3 Discussion

The metrics API can be considered in two parts:

- setting up the structure that represents what you want to track in terms of usage

- updating the usage tracking when a particular feature is used

The code shown above sets up the structure. Please ensure that your usage is consistent with the way other features are shown in the dialog.

Updating the usage so that the dialog shows a tick (check mark) rather than a cross obviously depends on the how the feature works. However a very common case is when the user makes a menu selection, and this is very easy to support by adding a `metricKey` entry to your menu definition (see Submenu cookbook sample for more on menus).

```
group.addChild({
  type: MenuItemType.ITEM,
  eventType: EventType.DO_ANOTHER_THING,
  label: 'Item 1',
  metricKey: Metrics.FIRST_THING,
  handler: handleItem
});
```

When the user selects that menu item, the corresponding metric entry will be set.

Some features are less suited to just triggering off a menu item, and a programmatic approach is better. For example, the `EXTRA_THING` key could be triggered by using code like:

```
OSMetrics.getInstance().updateMetric(Metrics.EXTRA_THING, 1);
```

In the example, this is triggered by either of the menu items (since the handler is shared). However a real usage would likely have it matching some specific user behaviour.

9.7.4 Full code

Listing 15: Metrics Cookbook Example - `src/index.js`

```
goog.declareModuleId('plugin.cookbook_metrics');

import OSMetrics from 'opensphere/src/os/metrics/metrics.js';

/**
 * @type {string}
 */
export const ID = 'cookbook_metrics';

/**
 * @type {string}
 */
export const MYGROUP = 'Cookbook Group';

/**
 * Our event types
 * @enum {string}
 */
export const EventType = {
  DO_SOMETHING: 'cookbook:do_y',
  DO_ANOTHER_THING: 'cookbook:do_x'
};

export const Metrics = {
  FIRST_THING: 'Metric for First Item',
  SECOND_THING: 'Metric for Second Item',
```

(continues on next page)

(continued from previous page)

```
  EXTRA_THING: 'Metric for programmatic item'
};

/**
 * Process a menu item
 * @param {os.ui.menu.MenuEvent} event The menu event.
 */
export const handleItem = function(event) {
  alert('item selected:' + event.type);
  OSMetrics.getInstance().updateMetric(Metrics.EXTRA_THING, 1);
};
```

Listing 16: Metrics Cookbook Example - `src/cookbookmetrics.js`

```
goog.declareModuleId('plugin.cookbook_metrics.CookbookMetrics');

import MetricsPlugin from 'opensphere/src/os/ui/metrics/metricsplugin.js';
import {Metrics} from './index.js';

/**
 */
export default class CookbookMetrics extends MetricsPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();

    this.setLabel('Cookbook');
    this.setIcon('fa fa-book');
    this.setCollapsed(true);
    this.setDescription('Plugin for metrics example.');

    const leaf = this.getLeafNode();
    this.addChild(leaf, {
      label: 'First Metric Item',
      description: 'This is an item.',
      key: Metrics.FIRST_THING
    });

    this.addChild(leaf, {
      label: 'Second Metric Item',
      description: 'Does something.',
      key: Metrics.SECOND_THING
    });

    this.addChild(leaf, {
      label: 'Third Metric Item',
      description: 'Combination, programmatic.',
      key: Metrics.EXTRA_THING
    });
  }
}
```

Listing 17: Metrics Cookbook Example - src/cookbookmetricsplugin.js

```
goog.declareModuleId('plugin.cookbook_metrics.CookbookMetricsPlugin');

import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import MenuItemType from 'opensphere/src/os/ui/menu/menuitemtype.js';
import * as spatial from 'opensphere/src/os/ui/menu/spatial.js';
import MetricsManager from 'opensphere/src/os/ui/metrics/metricsmanager.js';

import CookbookMetrics from './cookbookmetrics.js';
import {ID, MYGROUP, EventType, Metrics, handleItem} from './index.js';


/**
 * Cookbook example of metrics
 */
export default class CookbookMetricsPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;
  }

  /**
   * @inheritDoc
   */
  init() {
    const metricsManager = MetricsManager.getInstance();
    metricsManager.addMetricsPlugin(new CookbookMetrics());

    const menu = spatial.getMenu();
    if (menu) {
      const root = menu.getRoot();
      let group = root.find(MYGROUP);
      if (!group) {
        group = root.addChild({
          type: MenuItemType.GROUP,
          label: MYGROUP,
          tooltip: 'Added by cookbook metrics example'
        });
        group.addChild({
          type: MenuItemType.ITEM,
          eventType: EventType.DO_ANOTHER_THING,
          label: 'Item 1',
          metricKey: Metrics.FIRST_THING,
          handler: handleItem
        });
        group.addChild({
          type: MenuItemType.ITEM,
          eventType: EventType.DO_SOMETHING,
          label: 'Item 2',
          metricKey: Metrics.SECOND_THING,
```

(continues on next page)

(continued from previous page)

```
            handler: handleItem
          });
        }
      }
    }
}

// add the plugin to the application
PluginManager.getInstance().addPlugin(new CookbookMetricsPlugin());
```

9.8 Tracks

9.8.1 Problem

You have a data source that updates over time, such as a near-real time track feed.

Example of this could include:

- Automatic Dependent Surveillance - Broadcast (from aircraft)
- Automatic Identification System (from shipping)
- Vehicle tracking systems
- Radar
- Military tactical data links

9.8.2 Solution

Use the OpenSphere tracks plugin.

Your code (e.g. plugin) ensures that the tracks plugin is available:

Listing 18: Tracks Cookbook example - imports

```
import {addToTrack} from 'opensphere/src/os/track/track.js';
import {createAndAdd} from 'opensphere/src/plugin/track/track.js';
```

Your code needs to wait for the Places plugin to be available (fully loaded) before attempting to add the track:

Listing 19: Tracks Cookbook example - Places plugin initialisation

```
    const placesManager = PlacesManager.getInstance();
    if (placesManager.isLoaded()) {
      this.onPlacesLoaded();
    } else {
      placesManager.listenOnce(EventType.LOADED, this.onPlacesLoaded, false, this);
    }
```

You can then create a new track, which might be in response to an initial connection, or in response to a server update (e.g. over a WebSocket)

Listing 20: Tracks Cookbook example - Create Track

```
onPlacesLoaded() {
  const track = createAndAdd(/** @type {!CreateOptions} */({
    features: this.getFeatures_(),
    name: 'Cookbook track',
    color: '#00ff00'
  }));
}
```

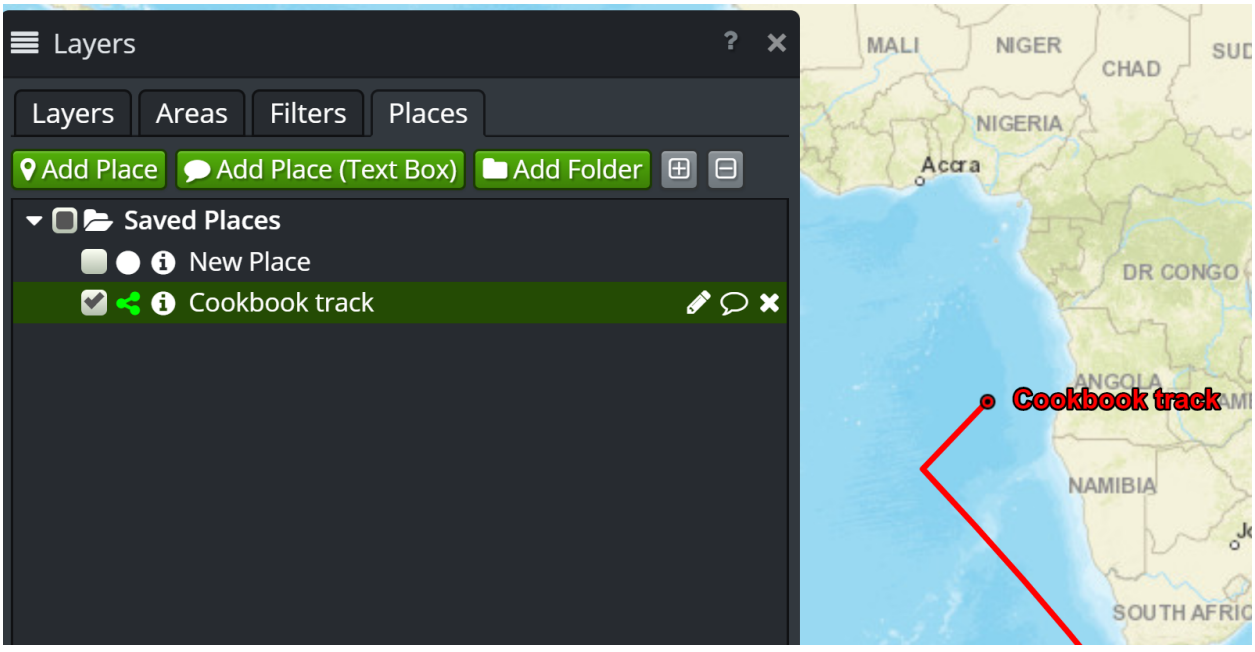
The track can then be updated with additional features in response to changes:

Listing 21: Tracks Cookbook example - Update Track

```
  addToTrack({
    features: this.getFeatures_(),
    track: track
  });
```

9.8.3 Discussion

The approach shown above will produce the track under the Saved Places layer (on the Places tab), as shown below:



You could create a separate track using `createTrack()` in place of `createAndAdd()`, and then add it to your own layer definition. See the KML parser implementation for an example of this.

Instead of using features, you can just pass coordinates instead, as shown below:

Listing 22: Tracks Cookbook Coordinates example - Create Track

```
  const track = createAndAdd(/** @type {!CreateOptions} */({
    coordinates: this.getCoordinates_(),
    name: 'Cookbook track',
```

(continues on next page)

(continued from previous page)

```
      color: '#00ff00'
    }));
```

Tip: In case you missed it, the CreateOptions object has a different key - `coordinates` in place of `features`. You pass exactly one of `coordinates` or `features`.

Similarly, you can pass coordinates to the update method as well:

Listing 23: Tracks Cookbook Coordinates example - Update Track

```
    addToTrack({
      coordinates: this.getCoordinates_(),
      track: track
    });
```

Tip: With both features and coordinates, you have to make sure your geometry is transformed into the map projection.

While your code could poll for updates, a streaming "server push" may be more appropriate in some scenarios. OpenSphere has two options for streams:

- `os.net.LongPoll`
- `goog.net.WebSocket`

If Web Sockets are supported, prefer `goog.net.WebSocket`.

`os.net.LongPoll` attempts to mimic the `goog.net.WebSocket` interface as much as possible. There may be other calls needed to setup/teardown streams depending on the remote server's API.

If neither the long poll or websocket options are supported (e.g. direct socket only), then it is not possible to connect from a web application such as OpenSphere. In this case, you will likely need a server proxy to adapt (e.g. "websockify") your streaming source.

9.8.4 Full code

To avoid the need for a server that provides updates, the example code makes periodic updates to the track position using `setInterval()` and the `modifyPosition_()` function. Those are artifacts of the example, and you wouldn't have that kind of function in your code.

Listing 24: Tracks Cookbook Features variation example - Full code

```
goog.declareModuleId('plugin.cookbook_tracks.TracksPlugin');

import EventType from 'opensphere/src/os/config/eventtype.js';
import RecordField from 'opensphere/src/os/data/recordfield.js';
import {PROJECTION} from 'opensphere/src/os/map/map.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import {EPSG4326} from 'opensphere/src/os/proj/proj.js';
import TimeInstant from 'opensphere/src/os/time/timeinstant.js';
import {addToTrack} from 'opensphere/src/os/track/track.js';
import PlacesManager from 'opensphere/src/plugin/places/placesmanager.js';
import {createAndAdd} from 'opensphere/src/plugin/track/track.js';
```

(continues on next page)

(continued from previous page)

```

const Feature = goog.require('ol.Feature');
const Point = goog.require('ol.geom.Point');
const olProj = goog.require('ol.proj');

const {CreateOptions} = goog.requireType('os.track');


let transformToMap;

/**
 * Provides a plugin cookbook example for track creation and update.
 */
export default class TracksPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;

    /**
     * @type {number}
     */
    this.lat = -35.0;

    /**
     * @type {number}
     */
    this.lon = 135.0;

    /**
     * @type {number}
     */
    this.latDelta = 0.1;

    /**
     * @type {number}
     */
    this.lonDelta = 0.1;
  }

  /**
   * @inheritDoc
   */
  init() {
    transformToMap = olProj.getTransform(EPSG4326, PROJECTION);
    const placesManager = PlacesManager.getInstance();
    if (placesManager.isLoaded()) {
      this.onPlacesLoaded();
    } else {
      placesManager.listenOnce(EventType.LOADED, this.onPlacesLoaded, false, this);
    }
  }

  /**
```

(continues on next page)

(continued from previous page)

```
   * @private
   */
  onPlacesLoaded() {
    const track = createAndAdd(/** @type {!CreateOptions} */({
      features: this.getFeatures_(),
      name: 'Cookbook track',
      color: '#00ff00'
    }));

    setInterval(() => {
      this.updateTrack(/** @type {!Feature} */ (track));
    }, 2000);
  }

  /**
   * @private
   * @return {!Array<!Feature>} features array for current location
   */
  getFeatures_() {
    const coordinate = transformToMap([this.lon, this.lat]);
    const point = new Point(coordinate);
    const feature = new Feature(point);
    feature.set(RecordField.TIME, new TimeInstant(Date.now()));
    const features = [feature];
    return features;
  }

  /**
   * Update the position and post the new track location.
   * @param {!Feature} track the track to update
   */
  updateTrack(track) {
    this.modifyPosition_();
    addToTrack({
      features: this.getFeatures_(),
      track: track
    });
  }

  /**
   * @private
   */
  modifyPosition_() {
    this.lat += this.latDelta;
    this.lon += this.lonDelta;
    if (this.lat > 50.0) {
      this.latDelta = -0.05;
    }
    if (this.lat < -50.0) {
      this.latDelta = 0.05;
    }
    if (this.lon >= 160.0) {
      this.lonDelta = -0.05;
    }
    if (this.lon < 0.0) {
      this.lonDelta = 0.05;
    }
```

(continues on next page)

(continued from previous page)

```
  }
}

/**
 * @type {string}
 */
const ID = 'cookbook_tracks';

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TracksPlugin());
```

If you'd prefer to use the coordinates approach, a complete example is shown below:

Listing 25: Tracks Cookbook Coordinates variation example - Full code

```
goog.declareModuleId('plugin.cookbook_tracks.TracksPlugin');

import EventType from 'opensphere/src/os/config/eventtype.js';
import {PROJECTION} from 'opensphere/src/os/map/map.js';
import AbstractPlugin from 'opensphere/src/os/plugin/abstractplugin.js';
import PluginManager from 'opensphere/src/os/plugin/pluginmanager.js';
import {EPSG4326} from 'opensphere/src/os/proj/proj.js';
import {addToTrack} from 'opensphere/src/os/track/track.js';
import PlacesManager from 'opensphere/src/plugin/places/placesmanager.js';
import {createAndAdd} from 'opensphere/src/plugin/track/track.js';

const olProj = goog.require('ol.proj');
const Feature = goog.requireType('ol.Feature');

const {CreateOptions} = goog.requireType('os.track');


let transformToMap;

/**
 * Provides a plugin cookbook example for track creation and update.
 */
export default class TracksPlugin extends AbstractPlugin {
  /**
   * Constructor.
   */
  constructor() {
    super();
    this.id = ID;
    this.errorMessage = null;

    /**
     * @type {number}
     */
    this.lat = -35.0;

    /**
     * @type {number}
     */
    this.lon = 135.0;

    /**
```

(continues on next page)

(continued from previous page)

```
     * @type {number}
     */
    this.latDelta = 0.1;

    /**
     * @type {number}
     */
    this.lonDelta = 0.1;
  }

  /**
   * @inheritDoc
   */
  init() {
    transformToMap = olProj.getTransform(EPSG4326, PROJECTION);
    const placesManager = PlacesManager.getInstance();
    if (placesManager.isLoaded()) {
      this.onPlacesLoaded();
    } else {
      placesManager.listenOnce(EventType.LOADED, this.onPlacesLoaded, false, this);
    }
  }

  /**
   * @private
   */
  onPlacesLoaded() {
    const track = createAndAdd(/** @type {!CreateOptions} */({
      coordinates: this.getCoordinates_(),
      name: 'Cookbook track',
      color: '#00ff00'
    }));

    setInterval(() => {
      this.updateTrack(/** @type {!Feature} */ (track));
    }, 2000);
  }

  /**
   * @private
   * @return {!Array<!Array<number>>} coordinates array for current location
   */
  getCoordinates_() {
    const coordinate = transformToMap([this.lon, this.lat]);
    coordinate.push(0);
    coordinate.push(Date.now());
    const coordinates = [coordinate];
    return coordinates;
  }

  /**
   * Update the position and post the new track location.
   * @param {!Feature} track the track to update
   */
  updateTrack(track) {
    this.modifyPosition_();
    addToTrack({
```

(continues on next page)

(continued from previous page)

```
      coordinates: this.getCoordinates_(),
      track: track
    });
  }

  /**
   * @private
   */
  modifyPosition_() {
    this.lat += this.latDelta;
    this.lon += this.lonDelta;
    if (this.lat > 50.0) {
      this.latDelta = -0.05;
    }
    if (this.lat < -50.0) {
      this.latDelta = 0.05;
    }
    if (this.lon >= 160.0) {
      this.lonDelta = -0.05;
    }
    if (this.lon < 0.0) {
      this.lonDelta = 0.05;
    }
  }
}

/**
 * @type {string}
 */
const ID = 'cookbook_tracks';

// add the plugin to the application
PluginManager.getInstance().addPlugin(new TracksPlugin());
```

9.9 External Javascript Libraries

9.9.1 Problem

You have a plugin that needs to load some external javascript libraries.

9.9.2 Solution

Add an `index.js` configuration file to your plugin, and reference that in `package.json`. This will be merged into the main OpenSphere index page by opensphere-build-index.

9.9.3 Discussion

`index.json` might look something like this, where a couple of video.js dependencies are loaded.

```
'use strict';
```

(continues on next page)

(continued from previous page)

```
const fs = require('fs');
const path = require('path');
const resolver = require('opensphere-build-resolver/utils');

// if opensphere isn't linked in node_modules, assume it's a sibling directory
const appPath = resolver.resolveModulePath('opensphere') || path.join(__dirname, '..',
↪ 'opensphere');
const versionFile = path.join(appPath, '.build', 'version');
const version = fs.readFileSync(versionFile, 'utf8').trim().replace(/.*\//, '');

module.exports = {
  basePath: __dirname,
  appPath: appPath,
  appVersion: version,
  distPath: path.join(appPath, 'dist', 'opensphere'),
  templates: [{
    // add video libraries to main index page
    id: 'index',
    // don't generate index files here, only resolve the resources
    skip: true,
    resources: [{
      source: resolver.resolveModulePath('video.js/dist', __dirname),
      target: 'vendor/video.js',
      scripts: ['video.min.js'],
      css: ['video-js.min.css']
    },
    {
      source: resolver.resolveModulePath('@videojs/http-streaming/dist', __dirname),
      target: 'vendor/video.js',
      scripts: ['videojs-http-streaming.min.js']
    }]
  }]
};
```

The `resources` array part of the templates can contain several entries (two shown here).

The resolver `resolveModulePath()` call will find the specified directory, and the specified `scripts` and `css` entries will be linked into OpenSphere.

You can also use `files` (instead of, or as well as, `scripts` and `css`) to support as-needed ('lazy') loading of scripts, or make additional files available. An example might look like:

```
{
  source: resolver.resolveModulePath('...', __dirname),
  target: '....',
  files: ['data.json', '*.+(gif|png)', 'extrafiles']
}
```

Those three entries in the `files` array select:

- the specified file (i.e. `data.json`)
- all files with either of the specified extensions (.gif and .png)
- the `extrafiles` directory including contents

The `package.json` part simply needs to provide the file name as part of the `build` properties:

```
{
  "name": "....",
  ....
  "build": {
    ...
    "index": "index.js",
    ...
  },
 ...
}
```