geojs Documentation

Release 0.10.1

Kitware, Inc.

July 19, 2016

Contents

1 Quick start guide **3**
- 1.1 Build dependencies . . . 3
- 1.2 Getting the source code . . . 3
- 1.3 Building the source . . . 3
- 1.4 Using the library . . . 4

2 User's guide **7**
- 2.1 Dependencies . . . 7
- 2.2 Software conventions . . . 7
- 2.3 Class overview . . . 9
- 2.4 Coordinate systems . . . 10
- 2.5 Coordinate transformation methods . . . 10

3 Developer's guide **13**
- 3.1 Code quality tests . . . 13
- 3.2 Headless browser testing . . . 13
- 3.3 Selenium testing . . . 14
- 3.4 Code coverage . . . 16

4 Testing infrastructure **17**
- 4.1 selenium_test . . . 17
- 4.2 midas_handler . . . 17
- 4.3 upload_test_cases . . . 19

5 Indices and tables **21**

GeoJS is a flexible library for all kinds of geospatial visualizations from traditional point markers to 3D climatological simulations. It is designed for displaying large datasets (over 100,000 points) leveraging the power of WebGL. The programming interface was inspired by the widely used d3 and allows the user to generate features from arbitrary data objects with custom accessors. The API also provides custom mouse events that mimick browser level events, but work with WebGL features and even through active layers.

See the growing list of examples for a live demonstration of GeoJS's features or go to our Github repository to start hacking. GeoJS is in active development and many components are still being refactored in preparation for a stable release. If you have any questions or comments, feel free to join us on our mailing list.

CHAPTER 1

Quick start guide

1.1 Build dependencies

The following software is required to build geojs from source:

- Git
- Node.js

In addition, the following python modules are recommended for development and testing of geojs.

- Python 2.7
- Make
- CMake
- Pillow
- Requests
- Selenium

1.2 Getting the source code

Get the latest geojs source code from our GitHub repository by issue this command in your terminal.

```
git clone https://github.com/OpenGeoscience/geojs.git
```

This will put all of the source code in a new directory called `geojs`. The GeoJS library is packaged together with another library vgl. Formally, this library was included as a git submodule. Currently, vgl is downloaded from npm to integrate better with workflows used by web projects.

1.3 Building the source

Inside the new `geojs` directory, you can simply run the following commands to install all dependent javascript libraries and bundle together everything that is needed.

```
npm install
npm run build
```

Compiled javascript libraries will be named `geo.min.js` and `geo.ext.min.js` in `dist/built`. The first file contains geojs and vgl bundled together with a number of dependent libraries. The second file contains d3. The bundled libraries are minified, but source maps are provided

1.4 Using the library

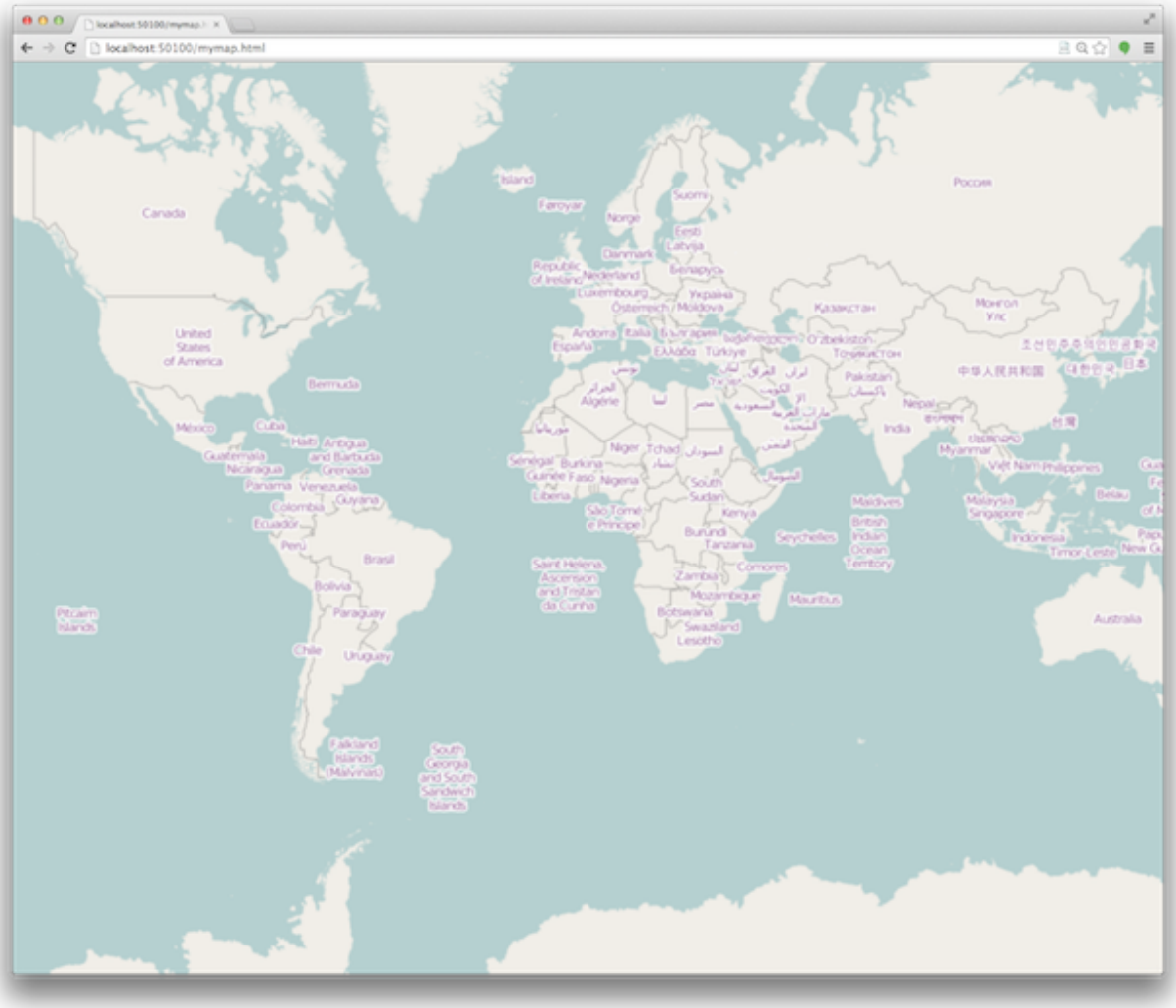
The following html gives an example of including all of the necessary files and creating a basic full map using the *osmLayer* class.

```
<head>
    <script charset="UTF-8" src="/built/geo.ext.min.js"></script>
    <script src="/built/geo.min.js"></script>

    <style>
        html, body, #map {
            margin: 0;
            width: 100%;
            height: 100%;
            overflow: hidden;
        }
    </style>

    <script>
    $(function () {
        geo.map({'node': '#map'}).createLayer('osm');
    });
    </script>
</head>
<body>
    <div id="map"></div>
</body>
```

You can save this page into a new file at `dist/mymap.html`. To view your new creation, start up a web server with the command

```
npm run examples
```

Now, if you open up http://localhost:8082/mymap.html in your favorite webgl enabled browser, you should see a map like the following:



Additionally, you will be able to see all of the built-in examples at http://localhost:8082/examples with the example server running.

CHAPTER 2

User's guide

2.1 Dependencies

GeoJS depends on several Javascript libraries that must be loaded prior to use as well as a few recommended libraries for optional features. As a convenience, we provide a bundle containing all required and optional dependencies in a single minified file. This bundle is built as `dist/built/geo.ext.min.js`. If you are just building a simple page out of GeoJS like in the *quick start guide*, this will probably work well; however, when using GeoJS as part of an application, you may need to customize the loading order or versions of the bundled applications. In this case, you may need to include the sources manually or bundle them yourself. The following is a list of libraries used by GeoJS.

Table 2.1: Internally bundled GeoJS dependencies

Library	Version	Component
proj4	2.3	Core
GL matrix	2.1	Core, GL renderer
earcut	2.1	GL polygon feature
jQuery	2.2	Core

Table 2.2: External GeoJS dependencies

Library	Version	Component
d3	3.5	D3 renderer, UI widgets

Note: JQuery is now included in the distributed bundle. Internally, this version will always be used and exposed as `geo.jQuery`. GeoJS will also set the global variable `window.$` if no other version is detected.

2.2 Software conventions

At it's core, GeoJS is an object oriented framework designed to be extended and customized. The inheritance mechanism used provides an isolated closure inside the constructor to maintain private methods and variables. Prototypal inheritance is performed by a helper method called `geo.inherit`. This method copies public methods declared on the parent class's prototype. In general, classes inside GeoJS do not declare methods on the class prototype. Instead, methods are typically bound to the instance inside the constructor. This provides access to the private scope. As a consequence, a class should always call its parent's constructor before extending the implementation.

Another convention used by GeoJS eliminates the need to use the `new` keyword when constructing a new instance. This is done by checking `this` of the current context. If it is not an instance of the current class, then the constructor is called again on a new object and the result is returned to the caller.

The conventions we use result in the following boilerplate code in every class definition inside GeoJS.

```
// New class, 'B', added to the geo module derived from class, 'A'.
geo.B = function (args) {

    // Constructors take a single object to hold options passed to each
    // constructor in the class hierarchy.  The default is usually an
    // empty object.
    args = args || {};

    // Here we handle calling the constructor again with a new object
    // when necessary.
    if (!(this instanceof geo.B)) {

        // Note: this will only happen in the constructor called by the
        // user directly, not by all the constructors in the hierarchy.
        return new geo.B(args);
    }

    // Call the parent class's constructor.
    geo.A.call(this, args);

    // Declare private variables and save overridden superclass methods.
    var m_this = this,
        s_func = this.func,
        m_var = 1;

    this.func = function () {

        // Call the super method.
        s_func();

        m_var += 1;
        return m_this;
    };

    return this;
};

// Static methods and variables can be added here.
geo.B.name = 'Class B';

// Initialize the class prototype.
geo.inherit(geo.B, geo.A);
```

Note:

- Variable naming conventions
  - The instance (`this`) is saved as `m_this`.
  - Super class methods are saved with the prefix `s_`.
  - Private variables are prefixed with `m_`.
- Methods beginning with _ are meant to be protected so they should only be called from within the class itself or by an inherited class.
- Use `m_this` to reference the instantiation inside public methods.

- Constructor options are passed inside a single object argument. Defaults should be used whenever possible.
- When possible, functions should return the class instance to support method chaining. This is particularly true for class property setters.
- In many cases, class methods return `null` to indicate an error.

2.3 Class overview

GeoJS is made up of the following core classes. Click on the link to go to the documentation for each of the classes.

geo.map The map object is attached to a DOM element and contains all visible layers and features.

geo.renderer A renderer is responsible for drawing geometries and images on the map. This is an abstract class which serves to define the minimal interface for a renderer. Not all features are available in all renderers, and an appropriate renderer must be selected for a layer based on the features that will be used. If a renderer is requested when creating a layer, and that renderer is not supported by the current installation, a fallback renderer may be used instead and a warning sent to the console. geo.gl.vglRenderer requires webGL support. geo.d3.d3Renderer requires the d3 library to be present.

geo.layer Layer objects are created by the map's `createLayer` method. This is an abstract class defining the interfaces required for all layers. Every layer must have a specific renderer. The following are useful layer implementations.

geo.featureLayer This is the primary container for features such as lines, points, etc.

geo.osmLayer This layer displays tiled imagery from an openstreetmaps compatible tile server.

geo.gui.uiLayer This layer contains user interface widgets that should generally be placed on top of all other layers.

geo.feature Feature objects are created by the featureLayers's `createFeature` method. Features are created from an arbitrary array of objects given by the `feature.data` method. Properties of the features can be given as constant values or as functional accessors into the provided data object. The styles provided are largely independent of the renderer used; however, some differences are necessary due to internal limitations. The following are feature types currently available.

- geo.pointFeature
- geo.lineFeature
- geo.pathFeature
- geo.graphFeature
- geo.vectorFeature

Note: Some features types are only available for specific renderers.

geo.gui.widget This is an abstract interface for creating widgets that the user can interact with.

- geo.gui.domWidget
- geo.gui.svgWidget
- geo.gui.sliderWidget
- geo.gui.legendWidget

geo.mapInteractor This class handles all mouse and keyboard events for the map. Users can customize the mouse and keyboard bindings through this class.

geo.fileReader This is an abstract class defining the interface for file readers. Currently, the only implemented reader is geo.jsonReader, which is an extendable geojson reader.

geo.clock The clock object is attached to the map and is resposible for maintaining a user definable concept of time. The clock can run, paused, and restarted. The clock triggers events on the map to synchronize animations.

The API documentation is in the process of being updated. You can always find the latest version at http://opengeoscience.github.io/geojs/apidocs/geo.html.

2.4 Coordinate systems

A major component of GeoJS's core library involves managing several coordinate systems that are used to keep layers aligned on the screen. The following conventions are used in GeoJS's documentation and codebase when refering to coordinates:

Latitude/longitude coordinates Expressed in degrees relative to the WGS84 datum as objects using keys `x` for longitude and `y` for latitude. Longitudes are assumed to be in the range `[-180, 180]`. Some map projections (such as the default `EPSG:3857`) are periodic in `x` and handle automatic wrapping of longitudes.

GCS coordinates Expressed in standard units (usually meters) as defined by Proj.4, which is used to perform coordinate transformations internally. The coordinate system `EPSG:4326` is equivalent to latitude/longitude coordinates described above. Points in these coordinate systems are given as an object with keys x and y providing the horizontal (left to right) and vertical (bottom to top) positions respectively. GCS coordinates have an optional `z` value that is `0` by default. The units of `z` should be expressed in the same units as `x` and `y`.

Display coordinates Expressed in units of pixels relative to the top-left corner of the current viewport from top to bottom.

World coordinates These are the coordinates used internally as coordinates of the 3D scene in much the sense as defined in 3D graphics. The world coordinates are a rescaled and translated version of the GCS coordinates so that the world coordinates of the current viewport is near `1` in each axis. This is done to provide well conditioned transformation matrices that can be used acurately in contexts of limited precision such as GL or CSS. In order to achieve this, the world coordinate system is dynamic at run time and will change as the user pans and zooms the map. By convention, the world coordinates are given relative to a dynamic "scale" and "origin". Changes to these values trigger events on the map that allow layers and features to respond and update their views as necessary.

Layer coordinates To allow flexibility for layer/renderer implementation, layers are allowed to use their own custom coordinate system via the functions `toLocal` and `fromLocal`. Features inside a layer should always pass coordinates through these methods to access the coordinates inside the layer's context.

Feature coordinates Features have a GCS property attached to them that should be taken to mean a geographic coordinate system for the data passed into the feature. For features such as points, coordinates are automatically transformed into the map's GCS by Proj.4, then transformed into world coordinates, and finally into layer coordinates before being passed to the layer's rendering methods.

2.5 Coordinate transformation methods

To facilitate uniform tranformation between the many coordinate systems used inside a map object, there are many available transformation methods provided in the core API. These methods vary from being useful to all users of the library to methods that are only relevant to developers interacting with low level renderers or wishing to optimize performance. The following is a list of transform methods present in the library as well as example uses for them.

geo.map.gcsToDisplay/displayToGcs(c, gcs) This is the most common tranformation method that converts from a geographic coordinate system into pixel coordinates on the map. If no GCS is given, the method will assume the coordinate system of the map. For example, to get the lat/lon of the point under the mouse you would get the pixel coordinates relative to the map's container and pass them to this method as `c` in `map.displayToGcs(c, 'EPSG:4326')`.

geo.map.gcsToWorld/worldToGcs(c, gcs) This performs the conversion to internal world coordinates that are scaled and translated to deal with round off errors. This method is made available so that layers can use a consistent base coordinate system from which the camera transforms are derived.

geo.layer.fromLocal/toLocal(c) This converts between world space and a custom coordinates system defined by each layer. The default implementation of these methods returns the original coordinate unmodified, but layers can choose to override this behavior as needed. Users generally do not need to call this method unless they are interacting with the low level context of the layer.

geo.camera.worldToDisplay/displayToWorld(c, width, height) This converts between world space coordinates and display pixel coordinates given a viewport size. In addition to thse methods, the camera class provides access to the raw transformation matrices for layers that can make use of them directly. For layers supporting CSS there is also a `camera.css` property that returns a CSS transform representing the current camera state.

CHAPTER 3

Developer's guide

Note: This guide assumes you have cloned and built the geojs repository according to the *Quick start guide*.

The selenium testing infrastructure of Geojs is run via CTest, it assumes that the testing "server" is started prior to execution. To start the server, just run

```
npm run start-test
```

This will start a server on the default port of `30100`. The port and selenium host names are configurable with cmake. For example inside the Kitware firewall, you can run the following to test on the selenium node on `garant`

```
cmake -DSELENIUM_TESTS=ON -DSELENIUM_HOST=garant /path/to/geojs
make
ctest -VV
```

You may need to also set the variable `TESTING_HOST` to your computer's IP address reachable by the selenium node.

Note: Typically, CMake is used to build outside of the source tree. This means you would create a new directory somewhare and point cmake to the geojs source directory. You may need to rerun `cmake` and `make` after making changes to your code for everything to build correctly. Try running `ccmake /path/to/geojs` for a full list of configuration options.

Geojs employs several different frameworks for unit testing. These frameworks have been designed to make it easy for developers to add more tests as new features are added to the api.

3.1 Code quality tests

All javascript source files included in the library for deployment are checked against ESLint for uniform styling and strict for common errors patterns. The style rules for geojs are located in the `.eslintrc` file in the root of the repository. These tests are preformed automatically for every file added to the build; no additional configuration is required. You can run a quick check of the code style outside of CMake by running `npm run lint`.

3.2 Headless browser testing

Geojs uses PhantomJS for headless browser testing of core utilities. Unfortunately because PhantomJS does not support webgl at this time, so code paths requiring gl must be either mocked or run via selenium.

The headless unit tests should be placed in the `tests/cases/` directory. All javascript files in this directory will be detected by the Karma test runner and executed automatically when you run `npm run test`. It is possible to debug these tests in a normal browser as well. Just run `npm run start` and browse to http://localhost:9876/debug.html. The test runner will automatically rebuild the tests as you modify files so there is no need to rerun this command unless you add a new file.

There are a number of utilities present in the file `tests/test-utils.js` that developers can use to make better unit tests. For example, a mocked vgl renderer can be used to hit code paths within gl rendered layers. There are also methods for mocking global methods like `requestAnimationFrame` to test complex, asynchronous code paths in a stable and repeatable manner. The Sinon testing library is also available to generate stubs, spies, and mocked methods. Because all tests share a global scope, they should be careful to clean up all mocking and instrumentation after running. Ideally, each test should be runnable independently and use jasmines `beforeEach` and `afterEach` methods for setup and tear down.

3.3 Selenium testing

Most tests for geojs require a full browser with webgl support. For these test, a framework based on Selenium is provided. This test framework is intentionally lightweight to allow for many different kinds of testing from simple Jasmine style unit tests to complicated mouse interactions with screenshot comparisons.

All selenium based tests should be placed inside subdirectories of `testing/test-cases/selenium-tests`. All subdirectories are assumed to be selenium tests by CMake and will be instrumented and run accordingly. Each subdirectory should, at a minimum, contain the following three files, which may be empty:

1. `include.css`: CSS that will be concatenated into a `style` node in the `head`.
2. `include.html`: HTML that will be concatenated into the `body`.
3. `include.js`: Javascript source that will be concatenated into a `script` node in the `head` after the inclusion of the geojs source and all dependent libraries.

Generally, developers are free to put arbitrary content into these files; however, one convention **must** be followed for the default instrumentation to work correctly. The javascript source should be wrapped in a global function called `startTest`. This function will be called automatically by the testing framework after all of the instrumentation is in place and the page is loaded. The `startTest` function will be called with function as an argument that should be called when page is ready to run the unit tests. This is provided as a convenience for the default behavior of `selenium_test.BaseTest.wait()` with no arguments. Developers can extend this behavior as necessary to provide more complicated use cases. As an example, see the `d3Animation` test case which sets a custom variable in a callback script for a test that is run asynchronously.

The compiled version of these tests are placed inside the deployment root so the users can manually see the test results. The path to each test is derived from the relative path inside `testing/test-cases/selenium-tests/`. For example, the test page in `testing/test-cases/selenium-tests/osmLayer/` is available at http://localhost:30100/test/selenium/osmLayer/ after starting the test web server.

The unit tests themselves are derived from Python's unittest module via a customized subclass `selenium_test.BaseTest`. Detailed documentation of the methods this class provides is given in the next section. Developers should feel free to extend this class with any generally useful methods as they become necessary for a wider variety test cases.

3.3.1 Example unit test

The following is a minimal example of a selenium unit test using the testing framework. More complicated examples can be found by examining the existing tests present in the source.

`hello/index.html`:

```
<div id="div-node"></div>
```

hello/index.css:

```
#div-node {
    text-align: center;
}
```

hello/index.js:

```
window.startTest = function (done) {
    $("#div-node").text("Hello, World!");
    done();
};
```

hello/testHelloWorld.py:

```
# Importing setupModule and tearDownModule will start up and
# shut down the web server automatically.
from selenium_test import FirefoxTest, setupModule, tearDownModule

# This test will run on firefox only.
class HelloWorld(FirefoxTest):
    testCase = ('hello', 'world')

    def test_main(self):
        # Resize the window to have consistent results.
        self.resizeWindow(640, 480)

        # Load the main html for this test directory.
        self.loadUrl('hello/index.html')

        # Wait for it to be loaded.
        self.wait()

        # Now we are ready to test the page.
        # The base class provide easy methods to test a screen shot.
        # This will take a screen shot and compare it against any
        # screenshots in the test image store at revision number 1.
        # Any failure here will raise an exception that will mark the
        # test as failed.
        self.screenshotTest('helloWorldScreenshot', revision=1)
```

3.3.2 Uploading screenshots to the image store

A script is provided in the source to help developers upload images to the data store in a way that they can be loaded automatically by the testing infrastructure. The script is built into `test/upload_test_cases.py` when selenium testing is enabled in CMake. When creating a new test (or updating a revision), the following is the recommended method for uploading test data for the example test `hello/` described above.

```
# inside the build directory
python test/upload_test_cases.py ../testing/test-cases/selenium-tests/hello
```

The script will run all the tests in this directory and prompt you if you want to upload a new image in the event that a screenshot test has failed. If you intend to start a new revision, then the revision number should be changed in the unit test source before running this script. Note: you must have write permission in the MIDAS GeoJS community before you can upload new images. Contact a community administrator for an invitation.

3.4 Code coverage

Code coverage information is generated automatically for all headless unit tests by Karma's test runner when running `npm run test`. The coverage information is submitted to codecov and cdash after every successful Travis run.

CHAPTER 4

Testing infrastructure

4.1 selenium_test

4.2 midas_handler

class midas_handler.**MidasHandler**(*MIDAS_BASE_URL='https://midas3.kitware.com/midas'*, *MIDAS_COMMUNITY='geojs'*)

Bases: object

Contains several utility function for interacting with MIDAS by wrapping api methods and caching the results.

community()

Get the id of the GeoJS community.

```
>>> midas.community()
{
    u'admingroup_id': u'121',
    u'can_join': u'1',
    u'community_id': u'40',
    u'creation': u'2014-06-02 11:38:38',
    u'description': u'',
    u'folder_id': u'11361',
    u'membergroup_id': u'123',
    u'moderatorgroup_id': u'122',
    u'name': u'GeoJS',
    u'privacy': u'0',
    u'uuid': u'538c9a7ead4a21c3b3e4e52724b3e6949487279edfad3',
    u'view': u'68'
}
```

Returns MIDAS response object.

Return type dict

getFolder(*name*, *root=None*)

Get a folder named name under root. If no root is given, use the community root.

```
>>> midas.getFolder('Testing')
u'11364'
>>> midas.getFolder('data', '11364')
u'11373'
```

Parameters

- **name** (*string*) – The folder name to find.
- **root** (*string*) – The id of the root folder.

Returns The id of the folder.

Return type string

Raises **Exception** – If the folder is not found.

getImages(*path*, *revision*)

Download images in an item at the given path and revision.

```
>>> .getImages(('Testing', 'test', 'selenium', 'osmLayer', 'firefox', 'osmDraw.png'), 2)
[<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=640x390 at 0x1019E9200>]
```

Parameters

- **path** (*tuple*) – The relative path from the community root.
- **revision** (*int*) – The item revision to download.

Returns List of Image.

Raises **Exception** – If the path or revision is not found.

getItem(*path*, *root=None*)

Get an item at the given path. If no root is specified, use the community root.

```
>>> midas.getItem(('Testing', 'data', 'cities.csv'))
{
    u'date_creation': u'2014-06-02 15:26:12',
    ...
    u'view': u'2'
}
```

Parameters

- **path** (*tuple*) – The relative path from `root`.
- **root** (*string*) – The id of the root folder.

Returns MIDAS response object

Return type dict

Raises **Exception** – If the item is not found.

getOrCreateItem(*path*)

Create an empty item at the given path if none exists otherwise return the item. This method will create folders as necessary while traversing the path.

Parameters **path** (*tuple*) – The relative path from the community root.

Returns MIDAS response object

Return type dict

login(*email=None*, *password=None*, *apiKey=None*)

Log into midas and return a token. If *email* or *password* are not provided, they must be entered in stdin. The token is cached internally, so the user will only be prompted once after a successful login. Alternatively, an apiKey can be provided as login credentials.

Parameters

- **email** (*string*) – The user's email address.
- **password** (*string*) – The user's password.
- **apiKey** (*string*) – The user's api key.

Return type string

Returns The login token.

uploadFile(*fileData*, *path*, *revision=None*)

Uploads a file to the midas server to the given path. If revision is not specified, it will create a new revision. Otherwise, append the file to the given revision number.

Parameters

- **fileData** (*string*) – The raw file contents to upload.
- **path** (*tuple*) – The relative path to the item.
- **revision** (*int*) – The revision number to append the file to.

Raises **Exception** – If the upload fails for any reason.

Returns MIDAS response object

Return type dict

4.3 upload_test_cases

CHAPTER 5

Indices and tables

- genindex
- search

C

community() (midas_handler.MidasHandler method), 17

G

getFolder() (midas_handler.MidasHandler method), 17
getImages() (midas_handler.MidasHandler method), 18
getItem() (midas_handler.MidasHandler method), 18
getOrCreateItem() (midas_handler.MidasHandler method), 18

L

login() (midas_handler.MidasHandler method), 18

M

MidasHandler (class in midas_handler), 17

U

uploadFile() (midas_handler.MidasHandler method), 19