HOGESCHOOL VAN AMSTERDAM

Uforia

Front-end developer guide

Uforia

This document serves as a guide for future developers who will work on the Uforia front-end interface. The document also describes the project setup and explains certain design/technology choices.

Introduction

This documents describes and explains some of design choices the Uforia team has made while working on the Front-end web interface of Uforia. This document is meant as a guide for developers that will continue work on the Uforia front-end interface in the future. A new developer should be able to understand the setup and basic design of the front-end so that he/she can continue further developing it without having to explore the entire project.

Table of contents

Uforia-Browser

Why NodeJS

The primary reason for using NodeJs was the ElasticSearchJS client. At the time of writing this is the most well documented, most feature rich and most up to date ElasticSearch (ES) client. The Javascript client plays very well with a NodeJS server, this approach is also recommended by the ElasticSearchJS documentation. Another advantage of using NodeJS is that it's very easy to set up and it require hardly any additional knowledge besides knowledge of the Javascript language.

Installation

Linux

To following steps will explain how to install NodeJS for Linux using the command line.

1. Install NodeJs using the commando '*Sudo apt-get install nodejs*'. This will install NodeJS on your system.
2. Create a directory where you'd like to install the server and move to that directory
3. Clone 'Uforia-browser' from github with the command 'git clone https://github.com/uforia/Uforia-browser.git' or download the latest version with the command 'wget https://github.com/uforia/Uforia-browser/archive/master.zip' and unpack it with 'unzip master.zip'.

For a more in depth install guide, view the install guide in the documents folder of the Uforia-Browser repository (https://github.com/uforia/Uforia-browser).

Windows

To be continued

OS X

To be continued

Configuration

To configure the NodeJS server you'll need to open 'uforia-browser.js' with an editor in the root directory of your installation. In the top section of this file there are certain variables e.g. host, port. You can change these variables to your liking so that the NodeJs server will run with a different configuration.

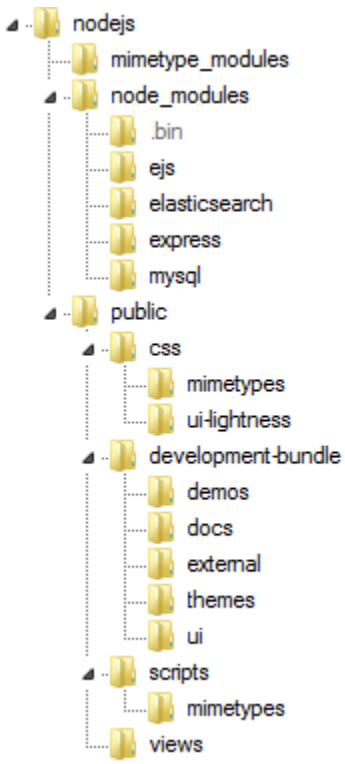
All setting listed at the top of the file are guaranteed to work properly when configured correctly. Any other changes may break the server entirely, be very careful if you choose to do this.

Uforia-browser.js

uforia-browser.js is the main file of the NodeJS server. It contains all the API calls, ElasticSearchJS requests, data manipulation functions and server configuration. If you need to change anything about the way the server works it should be changed/added/removed in this file.

Directory structure

The directory structure is as following:



'nodejs' is the root directory, this directory contains the other directories and the 'uforia-browser.js' file to run the server.

'mimetype_modules' contains files that are each a module for a mimetype. Every module provides functions that parses the data from ES in different ways, depending on what the user requested (e.g. data to create a graph or data to create a bar chart etc.)

'node_modules' contains all the necessary modules that the NodeJS server requires to run properly, under normal conditions you'll never have to touch this directory, except for changing permissions.

'public' everything in this directory will be served as static content by the NodeJS server. This folder contains everything from images to Javascript files to CSS files.

'css' contains all the CSS files, the global css for Uforia-Browser is stored in the root of this directory.

'css -> mimetypes' contains all the visualization specific CSS files. Every D3JS visualization has some specific CSS resources that it need, these will be loaded dynamically and are stored in this directory.

'css -> ui-lightness' contains the styles for the jQueryUI theme, this directory should not be touched.

'development-bundle' is a directory that belongs to the jQueryUI library, this directory should not be touched.

‘scripts’ contains all de Javascript files, the global Javascript file and the Javascript libraries for Uforia-Broswer are stored in the root of this directory.

‘scripts -> mimetypes’ contains all the visualization specific Javascript files. Every D3JS visualization has a specific Javascript files that actually runs the visualization, these will be loaded dynamically and are stored in the directory.

‘views’ contains the HTML files that define the basic structure of the pages.

NodeJS Modules

Express

Express (http://expressjs.com) is a web application framework which allows for developers to create a simple API in a matter of minutes. Express is used in Uforia-Browser to provide an API for browser client to interact with.

EJS

EJS (https://github.com/visionmedia/ejs) is an HTML templating engine for NodeJS. EJS and Express go together easily, currently EJS is only used to render HTML files served by Express.

ElasticSearchJs

elasticsearch.js (http://www.elasticsearch.org/guide/en/elasticsearch/client/javascript-api/current/) is the ES client for Javascript. It’s intended to work together with NodeJS (either remote or locally) and provides a very easy to use API to query ES.

Node MySQL

Node-MySQL (https://github.com/felixge/node-mysql) is a NodeJS library that provides an easy way for NodeJS to communicate with a database.

Python-shell

Python shell (https://github.com/extrabacon/python-shell) is a NodeJs library that provides an easy way to execute python scripts from a NodeJS instance and receive the results back via a callback function.

Node worker farm

Node worker farm (https://github.com/rvagg/node-worker-farm) is a NodeJs library that creates child node processes to handle work that would otherwise block the parent process for too long, heavy computations for example.

Front-end

Used libraries

D3.js

D3.js (http://d3js.org/) is the engine that powers all the data visualizations in Uforia-Browser. D3 visualizations are written in Javascript and in Uforia's case convert JSON data into visually appealing charts, diagrams, etc.

D3-tip

D3-tip (https://github.com/Caged/d3-tip) is a small library for the D3.js library. It allows for tooltip to be easily added to a D3 visualization.

jQuery

jQuery (http://jquery.com/) is well known Javascript library that makes it easy to manipulate elements in the DOM. In Uforia's case jQuery is mostly used to simplify tasks, like handling button clicks, adding and removing certain elements from pages.

jQuery-UI

jQuery (http://jqueryui.com/) adds UI elements to the jQuery library. jQueryUI makes it easy to add some basic elements like datepickers, progressbars, etc. At the time of writing it is only used to add a date picker to Uforia-brower.

Spin.js

Spin.js (http://fgnass.github.io/spin.js/) is a simple javascript library that allows for a highly customizable spinner to be added to a webpage.

Adding new features

Whenever you need to add a new visualization to Uforia there are certains steps you need to take, in the chapter the different steps you'll need to take a described in detail. It's important to follow these steps as described for the visualization to work properly and to retain the same structure as the rest of the project.

Adjusting the API

Adding the type and view

The NodeJS server keeps track of all the available types and the different views per type. If you add or remove a visualization you'll need to update the server as well. Adding or removing a view is a simple process, open the server file 'Uforia-browser.js' in a text editor and find the variables 'TYPES' and 'VIEWS' (they should amongst the constants at the top of the file). To add a new type, add an item to the 'TYPES' variable with the following format:

```
'mimetype'  :  'pretty name'
```

The mimetype has to correspond with the mapping that is used in Elasticsearch for that mimetype. So if you want to query the 'message_rfc822' mapping of elasticsearch the type should also be 'message_rfc822'

Each type can have multiple views (visualizations).To add a new view, add an item to the 'VIEWS' variable with the following format:

```
'mimetype' : { 'view_name' : 'view pretty name'  }
```

The mimetype should correspond with a value in the 'TYPES' variable. To have multiple view you can just add more entries to the mimetype object, for example:

```
'mimetype' : { 'view_name' : 'view pretty name', 'view_name2' : 'view2 pretty name' }
```

Parsing the data from ElasticSearch

Each visualization demands its data in a specific format, to accommodate for this there are so called mimetype modules. A mimetype module as a file that provides a set of public and private functions to parse data from elasticsearch in different ways. For each different type there is one mimetype module in the 'mimetype_modules' directory (see Directory structure for more info).

Mimetype modules ensure that the NodeJS server file doesn't become clogged with all kinds of visualization specific code. The basic structure of a mimetype module is given in appendix 1.

Create the D3 visualization Javascript file

To add a new D3 visualization you'll have to create a new Javascript file for the visualization. The naming and location of this file is important. The file will have to be placed in the 'mimetypes' directory of the

'scripts' directory (see Directory structure for more info). The filename should be the mimetype combined with a unique name that describes the kind of visualization it is. For example:

```
message_rfc822_graph.js
```

The 'message_rfc822' part is the mimetype, it will be a visualization for files that have been parsed with the message_rfc822 mimetype. The '_graph' part shows that this visualization will create a graph, the second part is needed so that multiple visualization per mimetype are allowed.

Once you've created the file you can go on to adding the code to the file. Each visualization file must have the function 'render(api_call)'. This function takes an URL as a parameter that, this URL is the call to the API which will be made up by what the user searched for in the interface. In most cases the visualization will work with JSON data, the URL is given to the 'd3.json()' function. This function requests the data from the API and lets you do something with it once it's retrieved. The basic structure of each visualization Javascript file is given in appendix 2.

This guide cannot help you with the actual creation of the visualization, a good source of inspiration for visualizations is the gallery provided by Mike Bostock, the creator of D3JS (https://github.com/mbostock/d3/wiki/Gallery) .

Create the D3 visualization CSS file

To give a visualization specific styles with CSS you'll have to add a separate CSS file for it. The naming and location of this file is important. The file will have to be placed in the 'mimetypes' directory of the 'css' direcotory (see Directory structure for more info). The filename should be the mimetype combined with a unique name that describes the kind of visualization it is. For example:

```
message_rfc822_graph.css
```

The 'message_rfc822' part is the mimetype, it will be a visualization for files that have been parsed with the message_rfc822 mimetype. The '_graph' part shows that this visualization will create a graph, the second part is needed so that multiple visualization per mimetype are allowed.

All the css classes of each visualization css file are currently prefixed with '#d3_visualization'. The visualization will always be shown in de div with the id 'd3_vizualization'. The prefix is done to ensure that the CSS code doesn't affect the style of the rest of the page. For example: a custom fontsize for a visualization will not affect the font size of the query input.

Once the file is created you can place all your visualization specific CSS in it. It will load automatically with the visualization.

API documentation

/api/search

Provides a way to query ElasticSearch via the API, currently both the query and the filter are a Boolean type which means that the querying and filtering works in terms of must (match) and must not (match).

parameters

Parameter	Additional info
type	A string that is the mimetype of the requested data (e.g. 'message_rfc822)
parameters	The search parameters together form a query to ElasticSearch. An Object that has two arrays: must and must_not, for must match and must not match. Each array has objects that follow have a field and query value. Field is the searched field and query is the search query for that field.
filters	The filters form a filter to filter the results of the ElasticSearch query. An Object that has two arrays: must and must_not, for must match and must not match. Each array has objects that follow have a field, start_date and end_date value. Field is the filtered field, start_date is the start of the date filter and end_date is the end of the date filter.
view	A string that specifies for what type of view the results should be rendered. For example: 'bar_chart' will manipulate the data to be compatible with a bar chart visualization in accordance with the type parameter
visualization	An object that contains the user set parameters for a specific visualization. For example: this object will have X and Y parameters for a bar chart visualization to specify what should be on the X and Y axis. The front end should ask the user for these paremeters.

/api/count

Provides a way to quickly count the number of results a query returns, this api call is identical to /api/search however it only counts the number of results, reducing the overhead and thus being faster and less resource intensive.

parameters

Parameter	Additional info
type	A string that is the mimetype of the requested data (e.g. 'message_rfc822)
parameters	The search parameters together form a query to ElasticSearch.

	An Object that has two arrays: must and must_not, for must match and must not match. Each array has objects that follow have a field and query value. Field is the searched field and query is the search query for that field.
filters	The filters form a filter to filter the results of the ElasticSearch query. An Object that has two arrays: must and must_not, for must match and must not match. Each array has objects that follow have a field, start_date and end_date value. Field is the filtered field, start_date is the start of the date filter and end_date is the end of the date filter.
view	A string that specifies for what type of view the results should be rendered. For example: 'bar_chart' will manipulate the data to be compatible with a bar chart visualization in accordance with the type parameter

/api/mapping_info

Returns every field in a mapping, including what type the field is

parameters

Parameter	Additional info
type	A string that is the mimetype of the requested data (e.g. 'message_rfc822)

/api/view_info

Returns the different types of view that are available for a mimetype. Normally this indicates which types of visualizations are available for a certain mimetype.

parameters

Parameter	Additional info
type	A string that is the mimetype of the requested data (e.g. 'message_rfc822)

/api/get_file_details

Returns every field in a mapping, including what type the field is

Takes parameters

Parameter	Additional info
type	A string that is the mimetype of the requested data (e.g.

	'message_rfc822)
tablename	A string that defines which table should be queried
hashids	An array with the hash ids of the files that will be returned

/api/get_types

Returns every type that is currently supported

Takes parameters

Parameter	Additional info
none	This api command doesn't take any paramters

Appendices

Appendix 2 – Mimetype module file skeleton

```
//imports
var util = require('./util');

//public functions
module.exports = {
    demoFunction: function(data) {
        print(data)
        //Do something with the data
        //...
        return newData;
    },

    demoFunction2: function(data){
        //Do something else
        return data;
    }
};

//Private functions
function print(){
    console.log(data);
}
```

Appendix 2 – d3 visualization file skeleton

```
//Every visualization file must have this function
function render(api_call){
    //Define variables
    ...
    //Call the API and handle results
    d3.json(api_call, function(error, result){
        stopSpinner(); //Stop the loading spinner
        //Handle errors
        if (error) {
            showMessage("An error occurred, please try another query");
        return console.error(error);
        }
        //Do something with the result from the API
        ...
    }
}
```