Development of a Modular JavaScript Data Display Framework

Simon Heimler[1]

[1]Faculty of Computer Science, University of Applied Sciences, Augsburg, 86161 GERMANY

Data visualization and representation is a very common task in the modern web and there are already a lot of specialized JavaScript libraries in existence. This paper proposes the concept and the development of a new JavaScript framework with the current working title "plastic.js". Instead of focusing on some specific visualization tasks it will provide a very general approach to (1) aggregating, (2) parsing and (3) displaying data. All those three main components are designed to be completely modular and easy to extend. As a consequence, plastic.js is not meant to replace existing visualization libraries, but to provide a modular platform to integrate them into a bigger framework. To the end-user, the plastic.js framework aims to provide an abstraction layer that contains all necessary information within a HTML tag that can be easily embedded or generated in the style of Web Components. The complexity of the visualization and the JavaScript logic should be hidden to the users by default. This creates a big benefit for them, since they don't have to research libraries, learn a scripting language and the usage of API's.

Index Terms: **Application programming interfaces, Data processing, Data visualization, Semantic Web, World Wide Web.**

I. Introduction

The idea for this project came in the context of working with Semantic MediaWiki (SMW) [1]. It is a common task to output data in various formats. SMW has already some default formats built in and there is an existing Result Format Extension [2] that provides additional formats. The problem is that those result formats are very tightly linked with the system around it. To write a new result format the developer has to know both about the client-side and the Server-side architecture and programming languages.
One example: The aggregation and parsing of data happens on the server side, the rendering of the result sometimes takes place on the client-side, sometimes not. The data-aggregation on the server side leads to another problem: Serving the site to the user is delayed until the data is queried, calculated and loaded, which can take a while depending on the complexity of the process.
So there is a need for a "cross-platform" Data Display framework. Ideally it should provide following features:

- It should work without having any dependency or knowledge about the outer system in which it is embedded in.
- It should have a modular architecture that allows easily extending the framework, even at runtime.
- It should provide a simple API that abstracts away most complexity. Ideally it should suffice to just provide the data and the options without having to write a single line of code.

My approach in solving those problems is the creation of plastic.js [3] which runs completely on the client-side browser and exposes an abstract and uniform API that hides most of the complexity from the user. If it is used within a CMS, the CMS just has to provide a thin wrapper around plastic.js that leverages the API. Many different CMSs could share the same framework and so the development effort.

At the time of writing plastic.js is in a working prototype phase.

II. Related Work

A. Generic Approaches

There are already a few projects that take the generic and broader approach to this problem.

1) Spark

The most similar project would be Spark [4] which also stems from the SMW Community. Sadly, it has been discontinued in 2012 and never made it beyond a prototype. But it shares the main idea of using the HTML Markup as a simple API and abstracts the complexity away very nicely. Spark doesn't support much result formats and also supports only RDF Data as input source.

2) Sgvizler

Sgvizler [5] uses a similar approach using HTML Markup to generate the Visualization. Like Spark it is limited to use only one category of Input Data, which is querying Data from an SPARQL Endpoint in this case. Sgvizler uses the Google Visualization API to generate the output.

3) Vega

An interesting and more alternative project is Vega [6]. It is generating Graphics from a single JSON file which contains both the data and the options in a machine readable format which serves as the abstraction layer. Vega uses D3.js to generate the graphics.

B. Visualization Libraries

It does make only limited sense to compare plastic.js to current visualization libraries, because they do not completely share the field of application. Rather plastic.js provides the bigger framework around those libraries. Still, most tasks that plastic.js performs are currently done by using visualization libraries.

Currently D3.js [7] is one of the most widely used and recognized Data Visualization Libraries. In fact, D3 stands for "Data Driven Documents", which suggest it's not just about visualization but about the representation of data in general.

This is one aspect plastic.js is going to adopt. D3.js also draws a clear separation between data (Model) and the actual rendered document (View). It is noteworthy that D3.js is a rather low-level visualization library. This leads to great flexibility but also requires the user to write actual code for achieving even very simple tasks.
D3.js is one of the libraries that plastic.js is using to generate visualizations.

III. Concept And Architecture

Below follows a brief introduction to the concept and architecture of plastic.js.

A. Programming Language

Since the framework should work independent from CMSs, it has to be developed in the one common programming language of the Web: JavaScript. The code will run solely on the client-side in the browser. Different server side applications could then just provide a thin wrapper that integrates plastic.js into the specific system.

B. API

To provide a simple API plastic.js uses existing HTML Elements and tags to enter the data and the options. Since the input tends to be bigger data or option files in JSON format, which usually have to be formatted with whitespace to keep readability, plastic.js reads them from script tags. This has the added advantages that code editors can cope with the data and options well and that the input data is declared by the correct mime-types.

This approach orients itself on some concepts of the emerging Web Components Technology [8] which uses native HTML Tags as an API / Abstraction Layer too. To implement a plastic element the user has to provide an embed code. An example is listed below.

```
<div id="table-ask-query" class="plastic-js"
style="height: 300px; width: 100%;">

    <script class="plastic-query" type="application/ask-
query" data-query-url="http://semwiki-
exp01.multimedia.hs-augsburg.de/ba-wiki/api.php">
        [[Category:Employee]]
        | ?Surename=Surename
        | ?Lastname=Lastname
    </script>

    <script class="plastic-options"
type="application/json">
        {
            "general": {
                "benchmark": true
            },
            "display": {
                "module": "simple-table",
                "options": {
                    "tableHead": true
                }
            }
        }
    </script>
</div>
```

C. Modular Architecture

As plastic.js should be easily extensible, a modular and scaling architecture [9] is needed. Currently there are three module types implemented: Query Modules, Data Modules and Display Modules. A module should only have access to the information that it needs to do its job. To provide loose coupling for the modules, the factory and facade pattern is used to instantiate new modules. Additionally there is a global and an element specific observer pattern to handle the asynchronous events and provide further decoupling [10]. With the current architecture it should be easy to implement an update mechanism that fetches new or additional data.

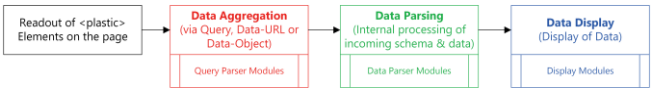

Figure 1: Modular structure from an (simplified) user perspective.

D. Asynchronous Architecture

1) Data loading

One big advantage of having all the code running on the client-side is that queries and data aggregation can run in separate and asynchronous processes. If there are multiple plastic elements on a page, they run in parallel and do not block themselves. That allows the site to load faster, especially in regards to the “felt” speed the user perceives.

2) Dependency Management

Since plastic.js can “host” several existing visualization libraries it has an asynchronously working dependency manager. If a module has external dependencies, plastic.js first aggregates them and lazy loads those which are actually needed. This keeps plastic.js small in size and even more modular.

IV. Implementation

A. JavaScript Style

Plastic.js is written to use the native prototype inheritance. Every plastic element tag in the HTML will receive a plastic element object instance. The modules are also written in an object oriented fashion. If no instances are needed, the code follows a simple singleton pattern.

B. Modular JavaScript

To provide modularity, plastic.js is divided into several files which are concatenated for production use in the build process.

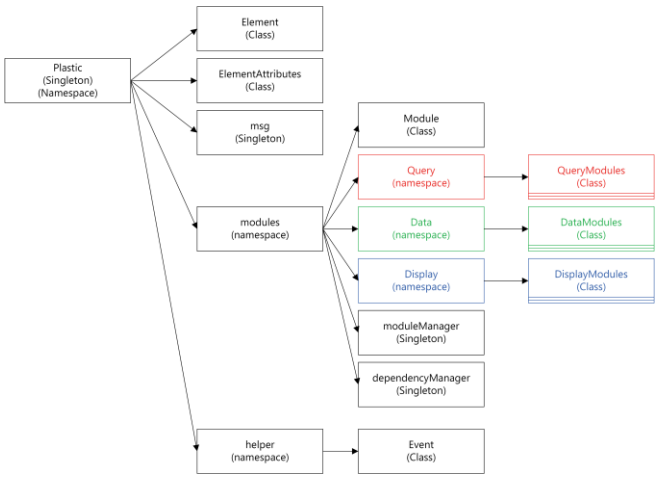

Figure 2 Project Structure

C. Challenges

1) Common Data format

One of the big challenges in developing this framework was to decide on an internal data format, which has to work with every incoming data type and also the "outgoing" display modules. This proves especially difficult since the incoming data could be in tabular structure, but also in a tree or even graph structure.

Since the graph structure is the most flexible one and can contain every other structure within, this seems to be a good choice for an all-purpose data structure. However this leads to choosing the most complex structure as the common denominator and complicates originally simple data structures significantly. The alternative would be to choose the simplest common data storage type which can be a simple table. RDF [11] has demonstrated that a complex graph can be stored as triples in a simple three column table.

The decision was made to go with the simplest possible data format that still allows for some flexibility. It consists of a table where each table cell is an array of zero or more Strings, Numbers or in some cases Objects. Objects provide further flexibility since they can represent more complex Entities like GeoCoordinates. But with the use of schemas – which will be described below - even a simple type like a string can be declared to be a date for example.

2) Usage of Data Schema

Often the incoming data has to be described so that a display module can interpret and render it correctly. Putting that information into the options would clutter them. It does make more sense to keep the data description near the data which it applies to.

Data Schemas are an elegant solution for this problem. They are written in a data format itself and contain additional information about the structure and semantics of the data. Schemas are sometimes even bundled with the data itself, which leads to self-describing data.

However there are many different approaches to schema formats with varied complexity. The one that showed most promise and the best tradeoff on flexibility and complexity is JSON Schema [12] together with the additional validation extension [13].

JSON Schema can describe both structure and basic semantics of a JSON File or JavaScript Object. There is an important distinction between data type and data format. The first declares the basic data type how the attribute is stored. The second describes the semantics and thus allows more sophisticated validation and interpretation of the value. JSON Schema is easily extensible so new custom formats and custom attributes can be added.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Example",
    "type": "object",
    "properties": {
        "name": {
            "type": "string"
        },
        "date": {
            "type": "string",
            "format": "date"
        }
    },
    "required": ["name", "date"]
}
```

JSON-LD [14] was also evaluated but it proved to be unnecessarily complex and more flexible than it has to be. It may be still interesting as an incoming Semantic Web data format.

Since the user should be able to write and provide data schemas it has to be a simple format that requires little to no prior knowledge about schema description. To provide this it is planned to have an even more simplified "data description" that is internally converted to a full JSON Schema object.

Some data sources include their schema by default. If this is the case the data module has to parse those specific schemas and convert them into the internal used schema format.

JSON Schema is also used internally to validate internal data structures. Modules can easily implement validation of the incoming data or options just by writing a validation object. Writing validation logic as code is purely optional.

3) Options

The user has to provide a lot of options how the data should be displayed. There are general settings which are shared between all display modules but also options that are specific to the display module that was chosen. Every module has to declare and validate which the options it requires.
It was decided to use JSON to input those options. They are – like the query – provided within a script tag.

D. Current project state

The architecture and internal ecosystem has been designed and implemented. There are two query modules and data modules

in existence which support querying and parsing data from SPARQL and ASK (SMW) APIs. Until now, however, only one display module is implemented yet, which supports simple table output.

V. Conclusion

At the time of the writing plastic.js is yet in a very early stage. The modular architecture has proved to be sufficient so far. However, since there are not many modules implemented yet it is hard to say if the architecture scales as well in the future. The most interesting issue is the internal data format. Will it work for all possible input and output formats that are to be implemented?

Since plastic.js is a prototype and work in progress this has still to be tested and evaluated. So far the project looks promising.

Acknowledgment

I would like to thank my advisor professor Wolfgang Kowarschick and the company Computer Bauer for making my research project possible with their support. I'd like to thank Yaron Koren for giving advice and contributing some ideas.

References

[1] SMW Community, *Semantic MediaWiki.* Available: http://semantic-mediawiki.org/ (2014, May. 21).

[2] Jeroen De Dauw, Yaron Koren, James Hong Kong, *Semantic Result Formats.* Available: https://semantic-mediawiki.org/wiki/Semantic_Result_Formats (2014, May. 19).

[3] Simon Heimler, Sebastian Huber, *plastic.js.* Available: https://github.com/Fannon/plastic.js (2014, May. 20).

[4] Denny Vrandečić, Andreas Harth, *Spark.* Available: http://km.aifb.kit.edu/sites/spark/.

[5] Martin G. Skjæveland, "Sgvizler: A JavaScript Wrapper for Easy Visualization of SPARQL Result Sets," *9th Extended Semantic Web Conference*, http://2012.eswc-conferences.org/sites/default/files/eswc2012_submission_303.pdf.

[6] Kanit Wongsuphasawat, *Vega: A Visualization Grammar.* Available: http://trifacta.github.io/vega/ (2014, May. 19).

[7] Michael Bostock, Vadim Ogievetsky and Jeffrey Heer, "D3: Data-Driven Documents," 2011.

[8] Dominic Cooney and Dimitri Glazkov, "Introduction to Web Components," W3C, 2013.

[9] Addy Osmani, *Patterns For Large-Scale JavaScript Application Architecture.* Available: http://addyosmani.com/largescalejavascript/ (2014, May. 19).

[10] A. Osmani, *Learning JavaScript design patterns*. Sebastopol, CA: O'Reilly Media, 2012.

[11] Guus Schreiber and Yves Raimond, "RDF 1.1 Primer," W3C, 2013.

[12] Francis Galiegue, Gary Court, *JSON Schema: core definitions and terminology.* Available: http://json-schema.org/latest/json-schema-core.html (2014, May. 19).

[13] ____, *JSON Schema: interactive and non interactive validation.* Available: http://json-schema.org/latest/json-schema-validation.html (2014, May. 19).

[14] Markus Lanthaler, Gregg Kellogg, and Manu Sporny, "JSON-LD 1.0," W3C, 2014.