x-project: a document-oriented toolkit to design and implement Web Applications based on HTML5 Web Components

Andrea D'Amelio
Università Roma Tre
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
damelio@ing.uniroma3.it

Enrico Marino
Università Roma Tre
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
marino@ing.uniroma3.it

Tiziano Sperati
Università Roma Tre
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
sperati@ing.uniroma3.it

Federico Spini
Università Roma Tre
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
spini@ing.uniroma3.it

ABSTRACT

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

CCS Concepts

•Information systems → Web applications; •Applied computing → Cartography; Format and notation; •Computer systems organization → *Client-server architectures; Real-time system architecture;*

1. INTRODUCTION

The need of automatize procedures to handle frequent activities involved in content publication on the web has set forth since Internet begining the success of Content Management Systems. Products like Joompla! or WordPress (which runs more than 23.3% of the top ten million websites (as of January 2015) [13]) are representative of a whole family of very successful systems that focus on the concepts of post and page, adapting just these two concepts to accomodate any further need of the user.

So emerged the need to handle arbitraty-schema data both inside a CMS or even in the more general context of web application.

To sustain this scenario, intense work and research around anatomy and operating of web applications have led to identify the operations that are identically performed by the (almost) totality of them. It is essentially the case of procedures related to user and session management, data access policies and CRUD method on basic data models. Software tools emerged nowadays (e.g. KeystoneJS or LoopBack) to automagically handle these operations once a description of the data types to deal with (i.e. model schemas) have been provided. This approach is perfectly suitable to speed up web applications development, mostly reling on external configuration files and less on procedural code [3].

Some of these tools also provide an auto-generated backend UI to interact with data: admin panels for data input, are available out of the box.

Relaing on these consideration, the main contribution of the work presented in this paper consists of the definition of a web development process driven by documents supported by a software toolkit, whose implementation and designing choices are also discussed. The web development cycle is a four-step process which can be recursively applied to all the web application views as well as to each their subcomponent. The toolkit consists of a library of Web Components which enable to realize a single page application by means of composition (and parametrization) of newly defined HTML5 tags.

Maximum development effort should be supplied to produce the schema definition documents. Data management UIs are provided automatically by the toolkit.

This kind of development extremize the concept of reuse

DocEng2015 Sep 8–11, 2015, Lausanne, Switzerland

DOI: 10.475/123_4

of code sice both UI components and model schema can be shared by different applications. Since the resulting code is a juxtaposition of tags, the complessive readability and maintainability of the produced code results dramaticaly increased.

The remainder of this document is organized as follows. In Section 2 we provide an overview of the proposed web development cyle process. Section 3 is devoted to describe the architecture and the tecnology stack exposed by applications developed with the toolkit, while section 4 presents the toolkit itself. Finally Section 5 reports about a case-study application: it is shown how to buil a CMS by mean of the toolkit introduced in this paper.

2. WEB DEVELOPMENT CYCLE

We model the web development process as a four-steps procedure that can be applyied recursively to each page (or view) of a web application as well as to every single complex component (or widgets) of the page itself.

This modellization is based on the reasonable assumption that server side operation on data models are nowadays be sufficently explored, and as proven by the *KeystoneJS* experience, at least one choice is available to automagically 1) generate server-side CRUD methods on models with ACL capabilities and 2) handle users and sessions, once a JSON description of data models and relations between them are provided to the system. This very JSON descriptor documents drive the whole process, actually composed by the following four steps.

1st step - JSON data model description. The JSON descriptors must be defined, specifying data type, relation, and user role read/write capabilities on particular portion of data.

2nd step - Model actions definion. Since CRUD operation coul not be enougth to describe all the needed operation further actions on models can be defined and exposed via http verbs.

3rd step - UI component definition. Then individual UI component can be defined, relying exclusively on CRUD operations and actions available on data models.

4th step - UI component assemplation. As last task, previously defined UI component have to be mounted to define application views. Assembly should be kept as simple as possible, in the case of x-project toolkit, it only consists of a juxtaposition of HTML5 tags.

3. ARCHITECTURE

Web applications developed exploiting `x-project` toolkit are full stack *JavaScript*.

On the server-side they rely on *Node.js*, exploiting the power of the *Loopback* framework by *Strongloop*. As mention above, the aim is to have a development process entirely document-driven, and those documents are the schemas of the models used by the application. These are JSON documents. Each document represents a model and presents the following fields: the `name` of the model, the set of `properties`, the list of `relations` to others models and the list of `ACL` (Access Control Layer) rules.

Loopback generates model's API from the models schemas, to let CRUD operations on models.

The API can be extended: the developer can add remote functions to models or add hooks to existing APIs to add behaviour before and/or after the API handler (to preprocess the request and/or postprocess the response).

The resulting API is RESTful, cookie free, signed by authentication token.

By default, applications have a built-in model that represent a user, with properties `username`, `email` and `password` for login and the property `role` used by the ACL module.

A very remarkable feature exposed by *Loopback* is that it abstracts from the particular DBMS utilized by the mean of an indirection layer, allowing to choose the preferred one, be it a noSQL or a graph-based one.

On the client-side, developed applications happen to be SPA (single page application) which exploit a variety of technologies, briefly described below.

Web Components are a collection of standards which are working their way through the W3C and landing in browsers at the moment. They allow to bundle markup and styles into custom HTML elements. *Custom Elements*[8], *HTML Imports*[9], *HTML Templates*[10], *Shadow DOM*[11]. Since these specifications are currently W3C Working Draft, they aren't fully supported across all major browsers, so As these technologies are implemented in browsers, the polyfills will shrink to gain the benefits of native implementations. [7]

Polymer library (https://www.polymer-project.org/) provides a thin layer of API on top of web components (native implementations and their polyfills) and several powerful features, such as custom events and delegation, mixins, accessors and component lifecycle functions, that makes it easier and faster to create Web Components. Similar to *Polymer* are *x-tag* and *Bosonic*. Web repositories http://component.kitchen and http://customelements.io already counts thousands of open source user-contributed custom elements.

4. X-PROJECT TOOLKIT

"Everything is an element", from an AJAX request to an entire web page. Every part of the website is encapsulated inside an element.

`x-project` provide a set of Polymer element for local routing, API requests, forms, lists, and style.

Elements can be customized through their attributes. We note that attributes could acts as inputs parameters (values that have effects to the element) or output parameters (values that are returned by the element); futhermore, the output parameter of an element could be an input parameter of another element.

The following elements performs local routing (for Single Page Application).

`<x-router>` implements local routing using *HTML5 Push State API*. It represent the core element of the app. It intercepts routes, create pages, and pass parameters to the page.

`<x-route>` represents a route-to-page mapping. Parameters presented in an URL are sent as attributes to the corresponding page.

```
1 | <x-route route="{{route}}" page="{{page}}">
```

`<x-link>` is an extension of the anchor element `<a>` that prevents the default behavior when a click event occurs,

blocking page request to the server and redirecting the request to the local router.

```
<a is="x-link" href="{{href}}">{{link}}</a>
```

The following elements handle HTTP RESTful API for the collections of the app.

`<api-collection-get>` gets a collection of models.

```
<api-collection-get name="{{name}}"
  where="{{where}}"
  page="{{page}}" perpage="{{perpage}}"
  items="{{items}}" schema="{{schema}}"
  count="{{count}}">
</api-collection-get>
```

Where `name` is the name of the collection to retrieve; `where` is object that specifies a set of logical conditions to match, similar to a `WHERE` clause in a SQL query; `page` and `perpage` are pagination parameters; `collection` is the retrieved collection; `schema` is the schema of the collection; `count` is the number of items in the collection (not the number of items retrieved, that corresponds to `perpage` value).

`<api-collection-post>` create a new model.

```
<api-collection-post name="{{name}}"
  model="{{model}}"></api-collection-post>
```

`<api-model-get>` retrieve a model.

```
<api-model-get name="{{name}}" model-id="{{id}}"
  model="{{model}}"></api-model-get>
```

Where `name` is the name of the collection of the model; `model-id` is the model id; `model` is the model retrieved (it acts as an output).

`<api-model-put>` update a model.

```
<api-model-put name="{{name}}" model-id="{{id}}"
  model="{{model}}"></api-model-put>
```

Where `model` is the model updated (it acts as an input).

`<api-model-del>` delete a model.

```
<api-model-del name="{{name}}" model-id="{{id}}">
</api-model-del>
```

`<api-filter>` generate dinamically (from a model schema) a form to create an API `where` clause filter.

```
<api-filter schema="{{schema}}"
  filter="{{filter}}"></api-filter>
```

The following elements are used to create forms.

`<x-input>` is an extension of the input element.

```
<x-input type="{{type}}" label="{{label}}"
  value="{{value}}"></x-input>
```

Where `type` can be `string`, `number`, `date`, `email`, `url`, `location` (with autocompletion based on Google Place API) and `file`.

`<x-form>` generate dinamically (from a model schema) a form to create/update a model.

```
<x-form schema="{{schema}}"
  model="{{model}}"></x-form>
```

The following elements are used to manage lists.

`<x-table>` generate dinamically (from a model schema) a table of models.

```
<x-table schema="{{schema}}" [editable]
  items="{{items}}"></x-table>
```

Where `schema` is used to generate the columns of the table; `collection` is used to generate the rows (the values) of the table; `editable` is a optional attribute.

`<x-pager>` generate the list of links to handle pagination.

```
<x-pager count="{{count}}" perpage="{{perpage}}"
  current="{{page}}"></x-pager>
```

Where `count` is the total number of item to paginate; `perpage` is the number of item per page; `current` is the current page clicked by the user.

By itself pagination doesn't paginate any list, but (as shown in the case study) it can be used in conjunction with `<api-collection-get>`, where the `current` output parameter of `<x-pager>` is the input `page` parameter of `<api-collection-get>`.

The style is based on `iron-flex-layout` [4], a CSS library of style mixins for cross-platform Flexible Box [12] layouts.

Even a page can be encapsulated in an element. `x-project` provides a set of pages for the admin part of the app, `<page-collection>` and `<page-model-edit>`, presented below.

5. CASE STUDY

In this section we discuss the design and the implementation of a blog platform.

For a blog platform the essential entities to model are: `Post` and `Tag`.

```
{
  "name": "Post",
  "properties": {
    "title": { "type": "string" },
    "posted": { "type": "date" },
    "content": { "type": "text" },
    "permalink": { "type": "string" }
  },
  "relations": [{
    "name": "tags",
    "type": "has_many",
    "model": "Tag"
  }]
}
```

```
{
  "name": "Tag",
  "properties": {
    "name": { "type": "string" }
  }
}
```

These models results in the following HTTP RESTful API (automatically generated by Loopback server).

```
GET|POST /api/Posts
GET|PUT|DELETE /api/Posts/:post_id
GET|POST /api/Tags
GET|PUT|DELETE /api/Tags/:tag_id
```

Since a snippet is worth a thousand words, in the following we present the pages of the app. It's important to note how easily a page can be built without writing code but assembling elements.

`index.html` imports the pages and set the router. `<page-collection>` and `<page-model-edit>` are provided by the toolkit.

```
<x-router>
  <x-route route="/admin/:collection"
    page="page-collection"></x-route>
  <x-route route="/admin/:collection/:model_id"
    page="page-model-edit"></x-route>
```

```
    <x-route route="/"
      page="page-posts"></x-route>
    <x-route route="posts/:id"
      page="page-post"></x-route>
</x-router>
```

`<page-collection>` shows models of a collection.

```
<template name="page-collection">
  <api-collection-get name="{{name}}"
    where="{{filter}}"
    page="{{page}}" perpage="{{perpage}}"
    items="{{items}}" schema="{{schema}}"
    count="{{count}}">
  </api-collection-get>
  <api-filter schema="{{schema}}"
    filter="{{filter}}"></api-filter>
  <x-table schema="schema" editable
    items="{{items}}"></x-table>
  <x-pager count="{{count}}" perpage="{{perpage}}
      "
    current="{{page}}"></x-pager>
</template>
```

`<page-model-edit>` shows the forms to update a model.

```
<template name="page-model-edit">
  <api-model-get name="{{collection}}"
    model-id="model_id"
    model="{{model}}" schema="{{schema}}">
  </api-model-get>
  <x-form schema="{{schema}}"
    model="{{model}}"></x-form>
  <api-model-put name="{{collection}}"
    model-id="{{model_id}}"></api-model-put>
</template>
```

`<page-posts>` show the list of posts.

```
<template name="page-posts">
  <api-collection-get name="Posts"
    page="{{page}}" perpage="10"
    collection="{{posts}}" count="{{count}}">
  </api-collection-get>
  <template is="dom-repeat" items="{{posts}}">
    <div>
      <h1>{{item.title}}</h1>
      <h3>{{item.posted}}</h3>
    <div>
  </template>
  <x-pager perpage="10" total="{{count}}"
    current="{{page}}"></x-pager>
</template>
```

`<page-post>` show a post. It is accessible via `/posts/:post_id` route. The `post_id` parameter is picked from the url by the router and passed to the page.

```
<template name="page-post">
  <api-model-get name="Posts"
    model-id="{{post_id}}" model="{{post}}">
  </api-model-get>
  <div>
    <h1>{{post.title}}</h1>
    <h2>by <span>{{post.author}}</span></h2>
    <h3>on <span>{{post.date}}</span></h3>
  </div>
  <div>{{post.content}}</div>
</template>
```

6. REFERENCES

[1] G. T. Heineman and W. T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] M. Nath and A. Arora. Content management system : Comparative case study. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 624–627, July 2010.

[3] V. Okanovic. Web application development with component frameworks. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*, pages 889–892, May 2014.

[4] A. Osmani and E. Bidelman. iron-elements. *Available at https://github.com/PolymerElements/iron-elements*, 2015. [Online; accessed 15-May-2015].

[5] S. Patel, V. Rathod, and S. Parikh. Joomla, drupal and wordpress - a statistical comparison of open source cms. In *Trendz in Information Sciences and Computing (TISC), 2011 3rd International Conference on*, pages 182–187, Dec 2011.

[6] A. Repenning, A. Ioannidou, M. Payton, W. Ye, and J. Roschelle. Using components for rapid distributed software development. *Software, IEEE*, 18(2):38–45, Mar 2001.

[7] Z. Rocha. Web components polyfills. *Available at http://webcomponents.org/polyfills/*, 2015. [Online; accessed 15-May-2015].

[8] W3C. Custom elements. *Available at http://www.w3.org/TR/custom-elements/*, 2014. [Online; accessed 15-May-2015].

[9] W3C. Html imports. *Available at http://www.w3.org/TR/html-imports/*, 2014. [Online; accessed 15-May-2015].

[10] W3C. Html templates. *Available at http://www.w3.org/TR/html-templates/*, 2014. [Online; accessed 15-May-2015].

[11] W3C. Shadow dom. *Available at http://www.w3.org/TR/shadow-dom/*, 2014. [Online; accessed 15-May-2015].

[12] W3C. Css flexbox. *Available at http://www.w3.org/TR/css3-flexbox/*, 2015. [Online; accessed 15-May-2015].

[13] W3Techs. Usage of content management systems for websites. *Available at http://w3techs.com/technologies/overview/content_management/all/*, 2015. [Online; accessed 15-May-2015].

[14] L. Wu, R. De Matta, and T. Lowe. Updating a modular product: How to set time to market and component quality. *Engineering Management, IEEE Transactions on*, 56(2):298–311, May 2009.