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

In this paper a software toolkit named `x-project` is introduced. It consists of a Web Component library which applied over a very powerful web framework, i.e. Loopback by Strongloop, realize an ibrid prototypal tool which bring togheter the customizability of a modern web framework with the ease of use of traditional CMSs.

Furthermore, the introduction of this tool implicitly defines a document-driven development process that extremize the concept of the reuse of code whose complessive readability, maintainability and extendibility result dramaticaly increased.

CCS Concepts

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

1. INTRODUCTION

Since the beginning of Internet, the ability to create and publish content on the web has made the success of Content Management Systems. Products like Joomla! or WordPress, born to handle simple website or blogs, are evolved to support web applications of any sort (e.g. personal portfolio, on-line newspaper, on-line shopping), running as of January 2015 more than 25% of the top ten million websites [6]. This evolution has been allowed by a plugin-based architecture, where each plugin is responsible to handle a functionality subset of the whole application, presenting the user whit a simple accessible configuration interface.

The large number of available plugins covers most of the common and frequently required customizations, thus avoiding to write ad-hoc code. Nevertheless, the implmentation of specific functional characteristics inevitably require to intervene at code level.

Where the effort required to add custom features to a CMS results too expensive, a web framework can be adopted instead. A web framework consists of a set of software facilities that aims to alleviate the overhead associate with common development activities. Web application coding effort, while eased by the web framework, is anyway rewarded by an increased level of extensibility and customizability of the produced application.

The most desiderable features for a wab frameork are: a) user management, b) session management, c) automatic generation of CRUD methods on data models exposed via HTTP RESTfull APIs, d) data access policies management. In order to effectively speed up web applications development, these facilities should be provided mostly reling on external configuration files and less on procedural code [1].

In this paper a software toolkit named `x-project` is introduced. It consists of a Web Component library which applied over a very powerful web framework, i.e. Loopback by Strongloop, realize an ibrid prototypal tool which bring togheter the customizability of a modern web framework with the ease of use of traditional CMSs.

Furthermore, the introduction of this tool implicitly defines a document-driven development process that extremize the concept of the reuse of code whose complessive readability, maintainability and extendibility result dramaticaly increased.

The remainder of this document is organized as follows. Section 2 is devoted to describe the architecture and the tecnology stack exposed by applications developed with the toolkit, while section 3 presents the toolkit itself. Section 4 reports about a case-study application: it is shown how to

DocEng2015 Sep 8–11, 2015, Lausanne, Switzerland

DOI: 10.475/123_4

buil a CMS by mean of the toolkit introduced in this paper. Finally section 5 outlines the development process implicitly defined by the introduction of the `x-project` toolkit.

2. ARCHITECTURE

A Web application developed exploiting the `x-project` toolkit, an `x-project` app, is a full stack *JavaScript* Single Page Application.

Server side.

On the server-side, an `x-project` app is based on:

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

Client side.

On the client-side, an `x-project` app is based on:

Web Components are an umbrella term for four different W3C specifications [5]: *Custom Elements* to define custom HTML elements; *HTML Templates* to define blocks of markup with the ability to inject dynamic content into; *Shadow DOM* to scope markup and styles in a separate DOM tree; *HTML Imports* to include and reuse HTML documents in other HTML documents. Each of these pieces is useful individually. But when combined, this whole package gives you: *Composability*, being able to create whole sites and apps by putting different elements together; *Encapsulation*, isolating markup, style, and behavior logic so they don't leak into the rest of the page; *Reusability*, extending existing elements to create new elements, allowing to stop reinventing the wheel. That means we can finally fill any gaps in the platform in a meaningful way. Which also opens an opportunity for a less fragmented ecosystem, where components can truly interoperate with each other. Since these specifications are currently W3C Working Draft, they aren't fully supported across all major browsers. `webcomponents.js` is a set of polyfills built on top of the Web Components specifications. As these technologies are implemented in browsers, the polyfills will shrink to gain the benefits of native implementations [3].

Polymer library (https://www.polymer-project.org/) provides a thin layer of API on top of Web Components 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.

3. 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). Values in parameters could be hard-coded (if they never change) or stored in variables. Different parameters in different elements could use the same variable, so, the value of an output parameter of an element could be used as input in an input parameter of another element.

Elements for local routing.

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.

```
<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>
```

Elements for API management.

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

`<api-collection-schema>` gets the schema of a collection.

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

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

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

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

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

Where: `name` is the name of the collection to retrieve; `where` is an object that specifies a set of logical conditions to match, similar to a `WHERE` clause in a SQL query; `page`

and `perpage` are parameters for the pagination; `items` are the retrieved models that match the query composed by the `where` and the pagination parameters; `count` is the size of the collection (the number of items of the collection).

`<api-collection-where>` generate dinamically (from a model schema) a form to create an API `where` clause filter. Specifically, for each property described in the model schema, it generates a corresponding input field.

```
<api-collection-where
  schema="{schema}"
  where="{where}" />
```

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

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

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 of a collection.

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

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

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

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

Elements for forms.

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}" />
```

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}" />
```

Elements for lists.

The following elements are used to manage lists.

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

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

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

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

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

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>`.

Elements for style.

The style is based on `iron-flex-layout` [2], a CSS library of style mixins for cross-platform Flexible Box [4] layouts.

Elements for admin pages.

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.

4. 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": {
    "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.

Admin part.

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

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

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

```
    where="{where}" />
  <x-table
    schema="{schema}"
    items="{items}" />
  <x-pager
    count="{count}" perpage="{perpage}"
    current="{page}" />
</template>
```

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

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

User part.

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

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

```
<template name="page-posts">
  <api-collection-get
    name="Posts"
    perpage="10" page="{page}"
    items="{posts}" count="{count}" />

  <template is="dom-repeat" items="{posts}">
    <li>{item.title} {item.posted}</li>
  </template>

  <x-pager
    perpage="10" total="{count}"
    current="{page}" />
</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="{id}"
    model="{post}" />

  <h1>{post.title}</h1>
  <h2>by {post.author}</h2>
  <h3>on {post.date}</h3>
  <div>{post.content}</div>
</template>
```

5. A 4-STEP WEB DEVELOPMENT PROCESS

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 as RESTful APIs via http verbs.

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

6. REFERENCES

[1] 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.

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

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

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

[5] W3C. Web components specs. *Available at https://github.com/w3c/webcomponents*, 2015. [Online; accessed 15-May-2015].

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