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

This work introduces the `x-project` toolkit, a software library essentially composed by a collection of Web Components realized via Polymer framework. The toolkit is then applied along with a modern web framework, namely Loopback by Strongloop, to realize an hybrid prototypal tool which brings together the customizability of a modern web framework with the ease of use of traditional CMSs.

Furthermore, the toolkit usage implicitly defines a document-driven development process that leads to a very readable, maintainable and extensible code by imposing a neat logic decomposition that strongly supports an engineered design of the web application.

CCS Concepts

•Software and its engineering → Development frameworks and environments; Software development techniques; •Information systems → *Web applications;*

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 (from personal portfolio, to on-line newspaper or on-line shopping), running as of January 2015 more than 25% of the top ten million websites [5]. This evolution has been allowed by a plug-in based architecture, where each plug-in is responsible to handle a functionality subset of the whole application, presenting the user with a simple accessible configuration and management interface.

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

When 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 with an increased level of extensibility and customizability of the resulting application.

The most desirable features for a web framework are: a) user management, b) session management, c) automatic generation of CRUD methods on data models exposed via HTTP RESTfull API, d) data access policies management. In order to effectively speed up web applications development, these facilities should be provided mostly relying 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, realizes an hybrid prototypal tool which brings together the customizability of a modern web framework with the ease of use of traditional CMSs.

Furthermore, the use of the toolkit alongside the web framework implicitly defines a document-driven development process that extremes the concept of reuse of code whose overall readability, maintainability and extendibility result dramatically increased.

The remainder of this document is organized as follows. Section 2 is devoted to describe the architecture and the technology stack exposed by applications developed with the toolkit, while section 3 presents the toolkit itself. Section 4

DocEng2015 Sep 8–11, 2015, Lausanne, Switzerland

DOI: 10.475/123_4

reports about a case-study application: it is shown how the toolkit can be used to build a blog. Finally section 5 summarizes the development process implicitly defined by 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 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 API to add custom behavior before and/or after the API handler (to pre-process the request and/or post-process 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. *Loopback* also introduces an indirection layer that allows to choose among (almost) any particular DBMS to use.

Client side.

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

Web Components are an umbrella term for four different W3C specifications [4]: *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 offers: *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. This brings 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 life-cycle functions, to facilitate the creation of 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, as listed below [1].

Elements can be customized through their attributes. Attributes can 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

[1] For the sake of conciseness, Polymer Elements are presented as empty elements, although empty element type is not supported. Furthermore, template variable are enclosed in single curly brackets while Polymer requires double curly brackets.

`where` clause and the pagination parameters; `count` is the size of the collection (the total number of items of the collection).

`<api-collection-where>` dynamically generates a form from a model schema, to create an API `where` clause filter. Specifically, for each property described in the model schema, it generates a corresponding input filter 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 auto-completion based on Google Place API) and `file`.

`<x-form>` dynamically generates a form from a model schema, to create/update a model.

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

Elements for lists.

The following elements are used to manage lists.

`<x-table>` dynamically generates a table of models from a model schema.

```
<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 items to paginate; `perpage` is the number of items per page; `current` is the current page selected by the user.

By itself pagination doesn't paginate any list, but it can be used in conjunction with `<api-collection-get>` (as shown in the case study), 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 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 the design and the implementation of a blog platform is presented.

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 the code of the pages of the app is shown. It is important to remark how easily a page can be built without writing code but assembling elements.

Admin part.

Tha admin part is composed by two pages: `page-collection` and `page-model-edit`. These pages are accessible via the following routes.

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

Where: the parameter `:collection` is the name of the collection to inspect; the parameter `:id` is the id of the model to edit. These parameters are set as attributes of the page element.

`<page-collection>` shows the 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>
```

Where: the value `collection` is picked from the url, via the parameter `:collection`; the value `schema` is the output of `<api-collection-schema>` and the input of `<api-collection-get>` and `<x-table>`; the value `items` is the output of `<api-collection-get>` and the input of `<x-table>`; the value `where` is the output of `<api-collection-where>` and the input of `<api-collection-get>`; the value `count` is the output of `<api-collection-get>` and the input of `<x-pager>`; the values `perpage` and `page` are the outputs of `<x-pager>` and the inputs of `<api-collection-get>`; every time the user (the admin) interacts with the pagination (`<x-pager>`) or the advanced search options (`<api-collection-where>`), `<api-collection-get>` regenerates the request to get the list of models using pagination and query parameters.

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

```
<template name="page-model-edit">
  <api-collection-schema name="{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="{id}" model="{model}" />
</template>
```

Where: the value `schema` is the output of `<api-collection-schema>` and the input of `<x-form>`; the value `model` is the output of `<api-model-get>` and `<x-form>` and the input of `<api-model-put>`. Once the page is ready (initialized and served by the local router): `<api-collection-schema>` fetch the `schema`; `<api-model-get>` fetch the model (a post or a tag) identified by `id`; `<x-form>` shows the form to edit the model. When the model changes (is updated via the form) `<api-model-put>` send a request to the server to update the database (via the corresponding HTTP RESTful API).

User part.

The user part is essentially composed by two pages: `page-posts` and `page-post`.

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

`<page-posts>` shows 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.date}</li>
  </template>
  <x-pager perpage="10" total="{count}"
    current="{page}" />
</template>
```

Where: the value `posts` is the output of `<api-collection-get>` and the input of the `<template>` iterator; for each `item` in `posts`, a list item with the post info (`title` and publishing `date`) is printed; `<x-pager>` component is the same used in the page `<page-collection>`.

`<page-post>` shows a post.

```
<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>
```

Where: the value `post` is the output of `<api-model-get>` and it is used to compose the page. Once `<api-model-get>` has fetched the `post` (identified by `id`), `title`, `author`, `date` and `content` of the `post` will be shown.

5. DOCUMENT-DRIVEN WEB DEVELOPMENT PROCESS

In the following the activities performed to build an application are outlined to the aim of elicit a development approach that imposes a neat logic decomposition that strongly supports an engineered design of the web application. The identified main activities can be arranged in the following four sequential steps.

1st step - JSON documents definition. The input JSON documents must be defined specifying data model schemas, model relations, user roles read/write capabilities on particular portion of data, and ancillary configurations (e.g. DBMS type). The output of this stage is a comprehensive set of HTTP RESTfull API to operate CRUD methods on defined data.

2nd step - Model actions definition. Further actions on models besides CRUD ones must be defined in this step and exposed as RESTful API via HTTP verbs.

3rd step - UI component definition. Distinct UI component must be defined, or retrieved from a collection of predefined components, configured and adapted. As concerns server communication, these components may avail only of the HTTP RESTfull API defined in the previous two steps.

4th step - UI component assembly. Distinct UI components are finally mounted to compose the application views. Assembly is kept as simple as possible: it only consists of a composition of HTML5 elements.

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

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