A Web Serverless Architecture for Buildings Modeling

Enrico Marino*, Danilo Salvati†, Federico Spini*, Christian Vadalà†
*Department of Engineering, Roma Tre University, Rome, Italy
Email:{marino,spini}@ing.uniroma3.it
†Department of Mathematics and Physics, Roma Tre University, Rome, Italy
Email:{salvati,vadala}@ing.uniroma3.it

***Abstract*—The motivations of relentless migration of software products toward services accessible via Web must be sought in the undeniable benefits in terms of accessibility, usability, maintainability and spreadability granted by the Web medium itself. It is the case of office suites, beforehand thought as resilient desktop applications, nowadays made available as Web applications, often equipped with real-time collaboration features and with no need for the user to explicitly install or upgrade them anymore. Although it could not be easy to envisage a Web-based graphic application due to its inherent complexity, after the recent and significant enrichment of the HTML5 APIs, a few first attempts appeared online in the form of vectorial drawing collaborative editors or VR oriented interior design environments. This paper introduces an effective Web architecture for buildings modeling that leverages the serverless pattern to dominate the developing complexity. The resulting front-end application, powered by Web Components and based on unidirectional data flow pattern, is extremely customizable and extendible by means the definition of plugins to augment the UI or the application functionalities. As regards the modeling approach, it offers (a) to model the building drawing the 2D plans and to navigate the building in a 3D first person point of view; (b) to collaborate in real-time, allowing to work simultaneously on different layers of the project; (c) to define and use new building elements, that are furnitures or architectural components (such as stairs, roofs, etc.), augmenting a ready to use catalog. This work suggests a path for the next-coming BIM online services, matching the professionals collaboration requirements typical of the BIM approach with the platform which supports them the most: the Web.**

1. Introduction

Nowadays we are seeing a relentless migration of software products toward services accessible via the Web medium. This is mainly due to the undeniable benefits in terms of accessibility, usability, maintainability and spreadability granted by the Web medium itself. Nevertheless these benefits don't come without a cost: performance and development complexity become major concerns in the Web environment.

In particular, due to the several introduced abstraction layers it is not always feasible to "port" a desktop application into the Web realm, an aspect to be taken into account even for the relevant hardware differences among all the devices equipped with a Web Browser. It can be even more arduous to tackle the inherent distributed software architecture (a client/server one at least) induced by the Web platform.

Nevertheless increasingly rich and complex Web applications began to appear, supported by the enriched HTML5 APIs, which thanks to the WebGL [1] (which enables direct access to GPU), Canvas [2] (2D raster APIs) and SVG [3] (vectorial drawing APIs), has paved the way for the entrance of Web Graphic Applications.

In this work we report about our endeavor toward the definition of a Web based buildings modeling tool which overcomes the aforementioned performance and development difficulties relying on a unidirectional data flow design pattern and on a serverless architecture [4], respectively.

A serverless architecture, on the contrary of what the name may suggest, actually employs many different specific servers, whose operation and maintenance don't burden on the project developer(s). These several servers can be seen as third party services (typically cloud-based) or functions executed into ephemeral containers (may only last for one invocation) to manage the internal state and server-side logic. Realtime interaction among users jointly working on the same modeling project, is for example achieved via a third party APIs for remote users collaboration.

The tool user interface, entirely based on **web components pattern**, has been kept as simple as possible: the user is required to interact mainly with two-dimensional symbolic placeholders representing parts of the building, thus avoiding complex 3D interactions. The modeling complexity is thus moved from the modeler to the developer which fills out an extendible **catalog** of customizable *building elements*. The modeler has only to select the required element, place and parametrize it according to the requirements. It is obvious that a large number of building elements has to be provided to ensure the fulfillment of the most modeling requirements.

The remainder of this document is organized as follows. Section 2 provides an overview of related work. Section 3 presents adopted architectural solutions. Section 4 contains

results, some conclusive remarks and figures future developments.

2. Related work

In this section we highlight some remarkable experiences aligned with the aim of our project. There are plenty of Desktop applications worth to be mentioned and analyzed, but in the following we deliberately focus on Web based works.

Shapespark[1] offers a web viewer of remarkable quality that allow the user to move inside a synthetic 3D indoor environment. Modeling phase is served in the form of plugins for different Desktop proprietary solutions.

Playcanvas[2] is a complete and powerful web based game creation platform which offers an integrated physical engine and a whole set of functionalities to support modeling. Although powerful and relatively simple to use, it doesn't focus on buildings modeling.

Floorplan[3] has been developed by Autodesk specifically for the architectural field, and for indoor renewal projects in particular. It is a 2D modeling tool which offer also a 3D walk-through mode.

Spini et al. [5] introduced a Web modeling and baking service for indoor environments. The modeling tools exposes a 3D interaction the user may not be accustomed to, an hitch we tried to outflank by avoiding 3D modeling interaction and let the user only face a "metaphoric" 2D interface.

3. Methodology

3.1. Application as Finite State Machine with centralized state

- unidirectional data flow
- How to change the state (actions and reducers)
- Immutability
- State as finite state machine
- Example of finite state machine

The application exploits the **unidirectional data flow pattern** to gain great benefits in terms of maintainability of the software objects, due to the lightness of the resulting architecture. As requested by the pattern, the software has to be based on a **centralized state**, which represents the application logic. Consequently, every feature has as its unique goal the obtainment of a new state coherent with the requested change, following various mechanisms based on **actions** (each one containing the data for the new state) and **reducers** (software objects that can write the new state) provided by the **Redux.js**[4] library. Usually, the unidirectional data flow pattern is associated with the **Immutability pattern**, which contributes to maintainability, avoiding side effects for state changes and providing smaller tests for application logic changes. We also benefits of undo/redo operations to an oldest/newer state by means of a replace of the current state with the previous/next one. As a consequence, the state can be seen as an **immutable tree structure** whose changes require: (i) The build of a new state `s'` equal to the previous state `s`; (ii) Execution of changes on the new state `s'`; (iii) Reference replacement from `s` to `s'`. Despite its simplicity, this pattern can nevertheless lead to memory waste, so we used the **Immutable.js**[5] library (made by facebook), which exploits structural sharing via hash maps tries and vector tries minimizing the need to copy or cache data. In our application, the state contains all the parts that there are on a plan, giving life to a finite state machine of all possible evolutions of the application state, where each node correspond to a **mode** (and consequently to a possible application state), and each edge correspond to an action from that point (which in our case is typically a javascript event such as `click` or `mousemove`). A part of the state machine is shown in figure 1. How we can see there are three nodes corresponding to three modes: (i) Node `idle`: waiting mode of the application. No action has been taken yet. (ii) Node `waiting_drawing_wall`: user has selected wall design tool, but he hasn't started the draw phase yet. (iii) Node `drawing_wall`: user has placed the starting point of the wall.

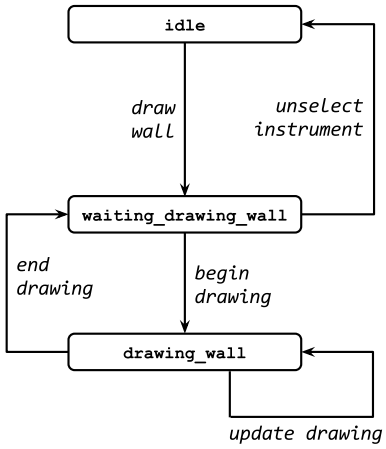

Figure 1. Subgraph of the state machine that show a wall creation.

3.2. Collaboration

- Review of related literature (Operational transformation)
- Central topology for collaboration

Now we will study the mechanisms chosen to provide collaboration for the same project, among users on different PCs. In literature there are many techniques for introducing collaboration between remote users. For example in [6]

1. https://www.shapespark.com/
2. https://playcanvas.com/
3. http://www.homestyler.com/floorplan/
4. http://redux.js.org/
5. https://facebook.github.io/immutable-js/

we can find the **Operational Transformation** technology, where a group of nodes exchanges messages without a *central control point*. It is based on two main properties: (i) you can transform changes relative to other changes and (ii) there is no matter in which order concurrent changes are applied, you end up with the same document.

However, the adoption of the unidirectional data flow pattern, with a global state, had suggested us to use another methodology for the collaboration feature. Following the principles of these patterns, the application is based upon a **centralized topology** for communication between nodes. With this approach the state is shared between several instances of the application with a communication infrastructure for the synchronization with the nodes. With this kind of topology, we avoided the peer-to-peer synchronization routines to remove implementation problems regarding the network infrastructure (for example ones caused by the *Network Address Translation*) common when using these kind of systems. Anyhow, this approach it does not influence the serverless nature of the architecture, as the communication systems are external (we adopted **Firebase**[6] for the data exchange between users)

3.3. Building elements

- Semplificazione modellazione
- Catalogo
- Proprietà elementi catalogo
- Quattro categorie di elementi

As we have discussed in section 2, modeling things can be difficult due to the 3D interactions required by tools. Consequently, we have tried to move the modeling complexity from the user to the developer, which is usually an expert of this field, with the creation of a **building element catalog**. In this way, one can add parts of the model simply the catalog is structured in categories and contains a list of **building elements**, each one representing an object we can add to our building. Each building element is represented using the following fields:

Name: the name of the building element chosen

2D rendering function: a function returning a 2D representation of this building element

3D rendering function: a function returning a 3D representation of this building element

Properties: the properties exposed by the building element and used by the user for customization

Moreover, the building element is associated with a **prototype**, which is the type of the element to represent. For our modeling purposes, we have identified four prototypes (so all representable elements can belong to only one of four fundamental types):

Walls, each kind of wall (perimeter, interior, structural) belongs to this category. The user can add a new wall selecting the start point and the end point of the element. The internal state of all the walls correspond to an undirected graph: each **node** represents a coordinate for an end of the wall, each **edge** match a specific wall. This internal representation helps us to provide a drag and drop interaction for this element type, in fact each drag is performed by means of a relocation od a node that cause, other that the displacement of the requested wall, a displacement of each adjacent wall.

Openings, each element that makes a hole in a wall (windows, doors, arches) belongs to this category. The user can add a new opening thanks to a snap on an existent wall. The openings internal state link each element with its wall.

Areas, each floor belong to this category. They are automatically generated thanks to an analysis of the wall graph. The algorithm is composed by the following phases: (i) Search of biconnected component by mean of Hopcroft-Tarjan algorithm (see [7]); (ii) Removal of edges that are not part of a biconnected component; (iii) Search of all cycles through an algorithm that do a double check of each edges sorted by angle; (iv) Search of maximal cycles correspondent to perimeter edges by an application of Gauss's area formula; (v) Removal of maximal cycles;

Objects, each element that can be arranged on an area is part of this category. The user can interact with the object and change its disposal and rotation.

3.4. UI components

- web components pattern
- Example of component: 2D viewer
- Example of component: 3D viewer
- diff and patch algorithm for 3D updates

The entire application has been developed using separated modules, following the **SOLID programming** principles, as it could be expanded. For the web part, we have used the **web components pattern**, with the implementation of **React.js**[7] written by Facebook. The main idea is to define the frontend application as a collection of components, each one rendered in a different way according with values assigned to the state. In Figure 2 se can see the user interface of the application with the most interesting components: the toolbar (where every button is another component), the sidebar and the 2D viewer:

we have an example of based on our application.

In particular, the most interesting components developed by us are the **viewers**, for example the 2D-viewer and the 3D-viewer. This component-based structure, allows us to define whatever state definition (for example in a tabular form) simply adding a new component.

The viewers structure is outlined in figure 3, where we can observe the great blocks. The first one is the *catalog* and as we have already seen in section 3.3 it contains all the building elements with their properties. We also see the

6. https://firebase.google.com/

7. https://github.com/facebook/react

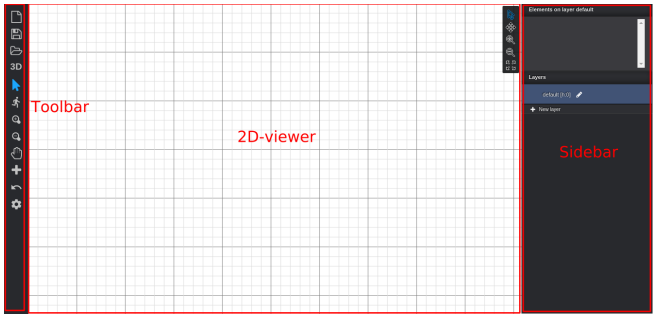

Figure 2. User interface of the application

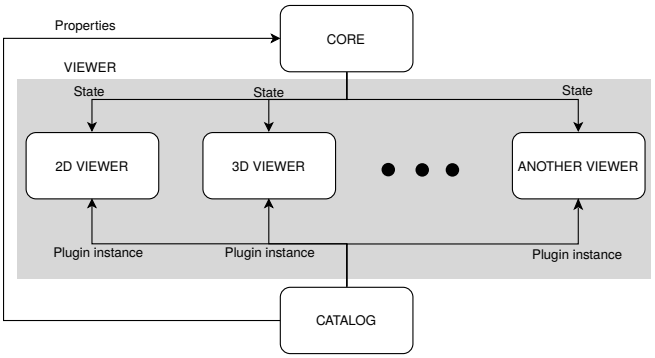

Figure 3. The architectural schema for the viewers

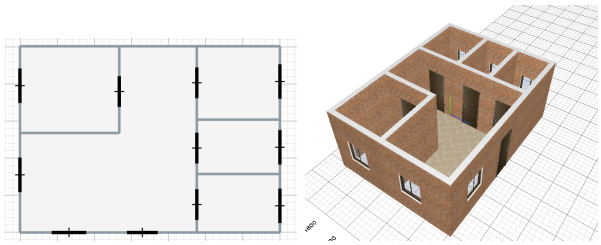

Figure 4. The 2D and 3D viewers for the state

application *core*, which manages the state and contains all drawing functionalities, and can instantiate several different viewers giving them the state for the representation. It communicates with the catalog taking the building models properties. At least we have the *viewers*, that are chosen observing the current state mode (see section 3.1). At the moment we implemented the 2D and 3D viewers (see figure 4)

2D Viewer. This viewer creates a 2D view of the building model. Given the state, it exploit the **Virtual DOM** (see [8]) implemented by React.js, to update only changed parts avoiding continuously global updates. All elements are rendered using SVG DOM tag. To perform pan and zoom operations, typically necessary in this kind of tool, we used ReactSVGPanZoom [8] a React component that adds pan and zoom features to SVG and helps to display big SVG images in a small space. It use affine matrix Math and event browser detection to adjust the image size to the required scale.

3D Viewer. It exploits the WEBGL-based modeling library **Three.js** [9] to build a 3D view of our building. To avoid global refreshes every time we update a part of the state, it has been implemented a `diff` and *patch* system standardized in [9]. So it has been implemented a data structure that maps the Three.js objects with the building elements inside the state. Every time the user launches a Redux.js action, the application compute the difference between the old state and the new one a redraw only the changed objects. In particular we can have the following *operations*: (i) **add**, (ii) **replace** and (iii) **remove**. For every operation it is determined a behavior based on the changed building element, changing also related building elements.

3.5. Serverless architecture

As already said in section 1, this project is built thinking at a **serverless** architecture, so it is mostly based on a frontend web application; this has the following advantages: (i) to avoid user hard installation or updating typically performed by not technical users; (ii) to offer a good abstraction level that makes the application platform and operation system independent; (iii) to allow a centralized new version deploy through a CDN (Content Delivery Network) update; (iv) to avoid a server overloading by moving heavy computations on the user client.

Now we can see all the components previously described from the serverless architecture point of view. For example in section 3.3, we have seen that to simplify modeling for the user we introduced a catalog, which is a FaaS (Function as a Service) worker that returns a 2D and 3D representation of a building element. Also the collaboration part is based on a FaaS, with the common state stored in an external DB provided by Firebase. Following this methodology we could also add user authentication features based on external BaaS providing all user management functionalities. In Figure 5 there is the architectural schema described above

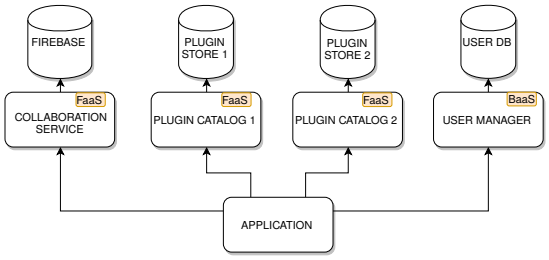

Figure 5. The serverless architecture for our application.

8. https://github.com/chrvadala/react-svg-pan-zoom
9. https://threejs.org/

4. Results

Now we can see the results obtained from the adoption of the techniques we have studied in the previous sections. The uniflow pattern allowed us to identify a consistent and global state which describes the state of the project and the business logic. Moreover, in conjunction with the adoption of a serverless architecture, it led to a slim architecture for the collaboration. Using the uniflow pattern with the Virtual DOM, permitted us to automatically render only changes to the state without complex optimization algorithms, like the one based on differences and patches used for the 3D rendering of the building model. In addition the system is extensible with the addition of geometric elements, in fact the user can register external services to add new ones. In conclusion, use of the Web Components allow us to expose our software in a modular way, improving maintainability.

4.1. Future work

The system could be expanded thanks to the modularity given by the serverless paradigm. As we already have a good user experience, to help the adoption of this application in other modeling contexts, we should improve the catalog adding new building elements

Acknowledgments

The authors would like to thank Stefano..... and Prof....

References

[1] D. Jackson, "WebGL Specification," Khronos, Khronos Recommendation, Oct. 2014, https://www.khronos.org/registry/webgl/specs/1.0.3/.

[2] J. Munro, J. Mann, I. Hickson, T. Wiltzius, and R. Cabanier, "HTML Canvas 2D Context," W3C, W3C Recommendation, Nov. 2015, http://www.w3.org/TR/2015/REC-2dcontext-20151119/.

[3] D. Jackson, E. Dahlström, J. Ferraiolo, A. Grasso, C. McCormack, P. Dengler, J. Fujisawa, D. Schepers, C. Lilley, and J. Watt, "Scalable Vector Graphics (SVG) 1.1 (Second Edition)," W3C, W3C Recommendation, Aug. 2011, http://www.w3.org/TR/2011/REC-SVG11-20110816/.

[4] M. Roberts, "Serverless Architectures." [Online]. Available: http://martinfowler.com/articles/serverless.html

[5] F. Spini, E. Marino, M. D'Antimi, E. Carra, and A. Paoluzzi, "Web 3D Indoor Authoring and VR Exploration via Texture Baking Service," in *Proceedings of the 21st International Conference on Web3D Technology*, ser. Web3D '16. New York, NY, USA: ACM, 2016, pp. 151–154. [Online]. Available: http://doi.acm.org/10.1145/2945292.2945309

[6] C. A. Ellis and S. J. Gibbs, "Concurrency Control in Groupware Systems," *SIGMOD Rec.*, vol. 18, no. 2, pp. 399–407, Jun. 1989. [Online]. Available: http://doi.acm.org/10.1145/66926.66963

[7] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973. [Online]. Available: http://doi.acm.org/10.1145/362248.362272

[8] J. Rotolo, "The Virtual DOM vs The DOM." [Online]. Available: http://revelry.co/the-virtual-dom

[9] P. Bryan and M. Nottingham, "JavaScript Object Notation (JSON) Patch," Internet Requests for Comments, RFC Editor, Tech. Rep. 6902, April 2013.