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 reports about the application user experience. Section 4 presents adopted architectural solutions. Finally, Section 5 contains some conclusive remarks.

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.

As regards support for users collaboration it worths to be mentioned the *Operational Transformation* (OT) approach [6]: a group of nodes exchanges messages without a central control point. Two main properties hold in this setup: (i) changes are relative to other user's changes (it works on "diffs") and (ii) no matter in which order concurrent changes are applied, the final document is the same. In our serverless architecture however, external (third parties) central synchronization point are allowed, making complexity introduced by protocols like OT less effective.

3. Application Experience

The application experience focus on supporting the user in a building modeling task. The exploited modeling approach requires the user to face as much as possible a bidimensional interface which allows her to define the floorplan and to place complex architectural elements (here called *building elements*) on it. Such *building elements* can be found in a pre-filled catalog, and when required can be further configured and customized through a side panel. This modeling approach move part of the complexity toward the developer of the customizable building elements, leaving to the final user the task to place and to configure the employed elements. A rich catalog of elements is thus crucial to answer to the users' modeling requirements.

Once the floorplan has been defined according to the *place-and-configure* approach, the system can automatically generate a 3D model which can be explored externally or in first person view, as shown in Figure **??**. Each *building element* in fact comprises either a *2D generating function* than a *3D generating function*, used to obtain models respectively used in the 2D floorplan definition and in 3D generated model.

The tool also has support for layers the user can exploit to organize her project, for example to group together semantically homogenous elements.

3.1. Building Elements

Along with the aforementioned 2D and 3D generating functions, an elements is fully specified by its univocal name and its properties, used by the user for customization. Each building element inherits from its *prototype* (one and only one). In the prototype are mapped both the inherent characteristics and user interactions needed to add the element to the project and/or configure it.

The catalog comprises then four different types of elements:

Lines. An element which belongs to this category is drawn selecting a start point and an end point. To move it one can drags one of the end point or can drags the entire line. An example: a *wall*.

Openings. An opening is an element that is linked to an element of the *line* type, making an hole on it. The user create a new opening by dragging on a chosen line. Examples are *doors* and *windows*.

Areas. An area is an element which is normally generated by insertion of the boundary vertices. For example a room basement is an area. In this case it is automatically generated from the lines representing walls, thanks to an algorithm which looks for closed cycles. 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.

Object. An element that is freely inserted into space with a drag and drop interaction is an object. Examples are tables and chairs.

3.2. User Interface

Figure 1 show the application user interface. It consists of the following components (from *left* to *right*):

Toolbar: it contains the button buttons mapping all operations the user can do in that context

1. https://www.shapespark.com/
2. https://playcanvas.com/
3. http://www.homestyler.com/floorplan/

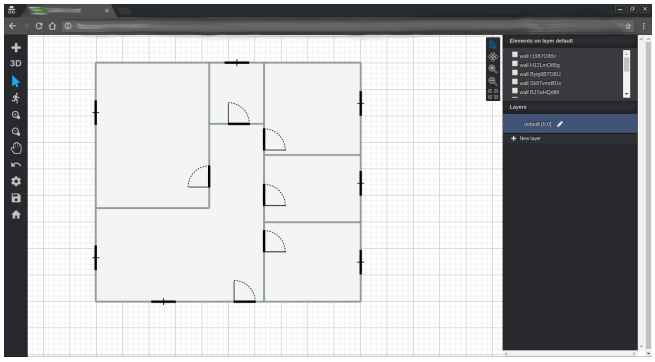

Figure 1. The user interface

Content area: it displays the main content requested by the user. In figure 1 the content area shows the 2D drawing area.

Sidebar: it contains all elements in the current layer, the layer list and the properties for the selected building element

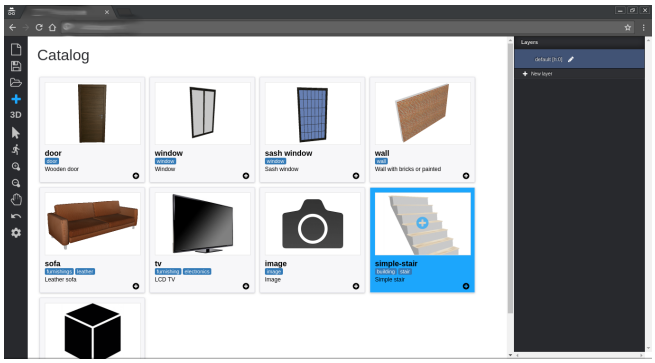

Figure 2. The building elements catalog

Figure 2 shows the building element catalog. When the user select one of the boxes, the application shows the 2D area and starts the interaction for the chosen building element.

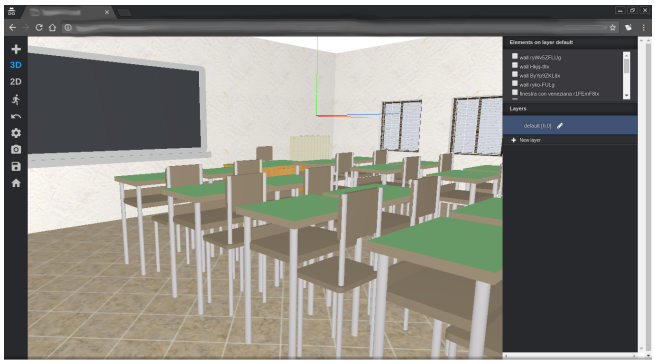

Figure 3. 3D visualization of a school

Figures 3 and 4 show some examples of 3D visualizations.

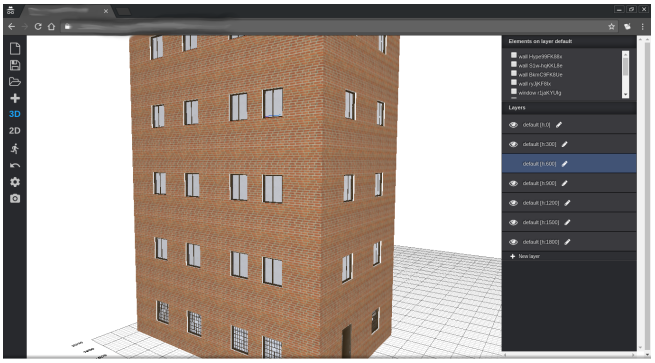

Figure 4. 3D visualization of a building

4. Serverless Architecture

According to [4], the deployed serverless architecture make extensive use of third parties services, or "*as-a-Service*-like" components, that replace ad-hoc software and hardware and fulfill the same tasks. In particular we have been able to delegate the following aspects to external and specialized services: application distribution, user management, model generation (heavy computation), user collaboration and state storage.

Distribution. To serve the application resources (JavaScript files, images, etc.) we rely on a keycdn CDN[4] (Content Delivery Network) and avoid any kind of webserver. This implies high availability and performance, but also represents a centralized method to upgrade client application without any user explicit action: once files have been updated on the CDN, at the subsequent reload of the web application, any user is provided with the updated version of the application.

User Management. User management activities, which requires a backend running code to accomplish user subscription and authentication, are delegated to Auth0[5], a Backend-as-a-Service (BaaS) specialized on accounting.

Model Generation. The generation of 3D models may require a while. In that case, especially on low-performing hardware, the computation can't be performed on the client-side, but rather on a powerful server which generate the model on demand, taking as input the parametrization made by the user through the web interface. On this purpose we chose to move this computation on AWS Lambda[6], a Function-as-a-Service (FaaS) on which runs the Python code responsible to generate 3D models, that automatically scale on the basis of the number of model generation tasks.

Users Collaboration. As regard the users collaboration, it requires some sort of synchronization among them. To use a single point of synchronization results in a much

4. https://www.keycdn.com/
5. https://auth0.com/
6. https://aws.amazon.com/lambda/

simpler architecture, so instead of relay on complex peer-to-peer architecture, which however would be a good choice aligned with the serverless paradigm we are pursuing, we opted for a synchronization mechanism based on Firebase[7], a Backend-as-a-Service working as a coordination manager. It broadcasts each change made by a user to all the others. Conflicts are avoided exploiting layers: each user lock the layer she is working on.

State Storage. The whole state of a modeling project is represented as a JSON document. To save and reload project state the user can easily downloads the corresponding JSON document, or, to remain "in the Cloud", the same document can be stored on a DataBase-as-a-Service (DBaaS), Orchestrate[8] for example. In this case we only relay on endpoints to save and to load a document: on a save event, client application serializes the state and passes it to the DBaaS which store it, ready to serve it on a load request issued at a later time. Once the client receives back the document, the state is automatically restored thanks to its reactive architecture.

The real support for serverless architecture is here not provided by the service that actually store the document, which as stated can be replaced even with a file download, but more precisely, by the software architecture that supports serialization and a reloading of the state. This architecture addresses two main concerns: (i) centralized immutable state and (ii) a reactive UI (i.e. modifications of the state reflect automatically on the user interface).

4.1. Centralized Application State

The application state is modeled using the data structure shown in Listing 1. It is essentially a collection of layers, each containing a collection of vertices, lines, areas and objects, each one of which is captured in a structure composed by: (i) information required by the object prototype; (ii) references mapping the relationship to other objects; (iii) metadata, namely the object customization entry point. Listings 2 and 3 give examples of data structures adopted to model a vertex and a line, respectively.

Information redundancies are exploited to decrease access times. Collections of objects are indexed by `id` thus allowing lookup in constant time, The `selected` field of each layer, whose contained information is already present in each object, grants direct access to selected elements, without searching.

Unidirectional data flow. The described data structure represents the centralized state required by the *unidirectional data flow pattern* [**?**] exploited by the application via *Redux.js*[9] library. The pattern prescribes that the state may be modified only by specific actors, called *reducers*, whose action is triggered by specific *actions* which contain

```
{
  "width": 3000, // canvas width
  "height": 2000, // canvas height
  "unit": "cm",  // unit of measurement
  "selectedLayer": "layer-1", // current layer
  "layers": {
    "layer-1": {
      "name": "default",
      "id": "layer-1",
      "altitude": 0,
      "opacity": 1,
      "visible": true,
      "vertices": {
        "HJAe59YF8Ux": {...}
        // ...
      },
      "lines": {
        "Hype99FK88x": {...}
        // ...
      },
      "openings": {
        "r1jaKYUIg": {...}
        // ...
      },
      "areas": {
        "Byg1oFKUIe": {...}
        // ...
      },
      "items": {
        "rkKU89U8e": {...}
        // ...
      },
     //selected element
      "selected": {
        "vertices": [],
        "lines": [],
        "openings": [],
        "areas": [],
        "objects": ["rkKU89U8e"]
      }
    }
  }
}
```

Listing 1: JSON serialized state, overall structure

```
{
  "id": "HJAe59YF8Ux",
  "x": 201,
  "y": 891,
  "prototype": "vertices",
  "selected": false,
  "lines": ["Hype99FK88x", "S1w-hqKKL8e"],
  "areas": ["Byg1oFKUIe"]
}
```

Listing 2: JSON serialized state, `vertex` structure

all the information needed by each reducer to accomplish the state change. Each application feature has to be implemented therefore as a couple of well isolated pieces of code (*action* / *reducer*). Preliminary experiments[10] on 2D drawing tools, in fact, highlighted the development complexity of an application of this kind in terms of large internal state modified by several user interactions, which was to be listened to and applied, resulting in a high coupling level between

7. https://firebase.google.com/
8. https://orchestrate.io/
9. http://redux.js.org/

10. https://github.com/cvdlab/walle

```
{
  "id": "Hype99FK88x",
  "type": "linear",
  "prototype": "lines",
  "vertices": ["HJAe59YF8Ux", "r11Z59tKIUg"],
  "openings": ["r1jaKYUIg", "BJVZ2M9FIIx"],
  "selected": false,
  "properties": {
    "height": 300,
    "thickness": 20,
    "coverA": "bricks",
    "coverB": "bricks"
 }
```

Listing 3: JSON serialized state, `line` structure

application logic and user interface. In our setup instead we defined a *state engine*, which represents the application logic, comprising actions and reducers, and encapsulates the centralized state. On this layer can transparently relay different interfaces.

Immutability pattern. *Immutability pattern* [**?**] is also applied, to avoid side effects on state changes performed by reducers. The state can be seen as an immutable tree structure whose changes are applied as follows: (i) clone the previous state `s` obtaining a new state `s'`; (ii) apply changes on the cloned state `s'`; (iii) Update reference from `s` to `s'`. It is worth noting that this approach provides out-of-the-box support for undo/redo operations: an oldest/newer state can be restored by means of a replacement of the current state with the previous/next one.

Despite its simplicity, this pattern can nevertheless lead to memory waste, due to the several copies of the state that must be held in memory. We addressed this issue using **Immutable.js**[11], a library which exploits structural sharing via hash maps tries and vector tries, thus minimizing the need to copy or cache data.

User interaction. We modeled the user interaction as a *Finite State Machine* (FSM) where each node corresponds to a *mode* (i.e. to a possible application state), and each edge corresponds to an action (e.g. a user interaction) that can be set off in the current state, (typically a JavaScript event mapping a user interaction such as `click` or `mousemove`).

Figure 5 shows the FSM relative to the wall drawing interaction. 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.

4.2. Reactive Component Based UI

The UI has been developed following the *Web Components pattern* [**?**], supported by *React.js*[12] framework.

11. https://facebook.github.io/immutable-js/
12. https://github.com/facebook/react

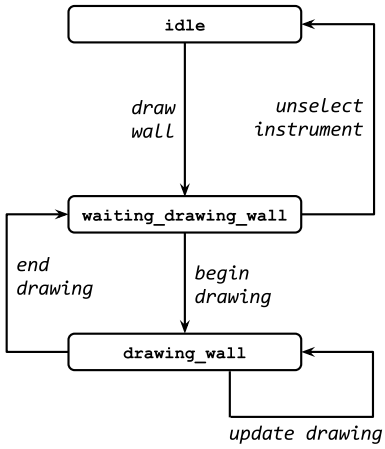

Figure 5. Subgraph of the state machine that show a wall creation.

The main idea is to define the frontend application as a collection of independent components, each one referencing a specific subset of the centralized state and able to render itself according to the actual values of that portion of the state. Web Components spawn from for high level generic containers, like the toolbar or the catalog, to very fine grained ones, buttons for example. The most interesting are the viewers of the building model: the *2D-viewer* and the *3D-viewer*.

Viewers. A viewer is a pivotal component since it shows the building model and allows user interaction with it. We built a *2D-viewer* and a *3D-viewer*.

The *2D-viewer* invokes the *2Dgf* of the building elements added to the model and renders its output using SVG elements. To cope with frequent updates coming from the user drawing interaction, it exploits the *Virtual DOM* [8], which permits to update only the modified part thus avoiding complete redrawing of the scene. To perform pan and zoom operations, typically necessary in this kind of tool, we develop an ad-hoc React component named *ReactSVGPanZoom*[13].

The latter offers two interaction ways: a 3D model can be inspected from the outside as well as walking inside it in a first-person point-of-view.

The *3D-viewer* invokes the *3Dgf* of the building elements added to the model and renders its output using WebGL primitives via **Three.js** [14]. Iit has been implemented a `diff` and *patch* system, standardized in [9]: Three.js objects are associated with building elements inside the state, so every time the user triggers an action that results in a state alteration, the application compute the difference between the old state and the new one a thus change only the affected object. In particular we can have the following *operations*: (i) **add**, (ii) **replace** and (iii) **remove**.

Figure 6 shows the interaction among viewers, state-engine and catalog. A viewer reacts to any state change

13. https://github.com/chrvadala/react-svg-pan-zoom
14. https://threejs.org/

updating its internal state and displaying the changes applied to the model. The generating function are pulled from the catalog where a descriptor for each building elements can be found.

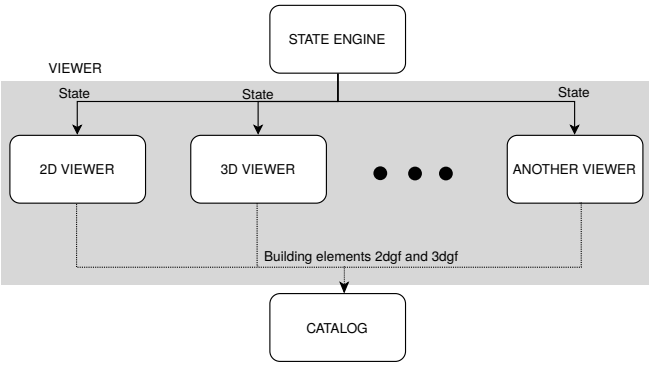

Figure 6. Viewers architectural scheme

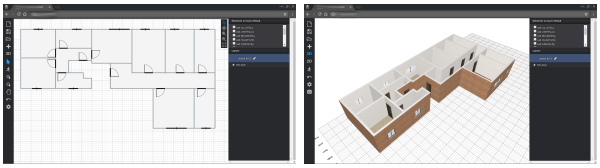

Figure 7. The 2D and 3D viewers for the state

5. Conclusions

In this work we outlined a serverless architecture to support buildings modeling in a Web environment. The serverless architecture that gives benefits in terms of availability, reliability, scalability, easiness of deployment, maintainability and upgradability is obtained by implementing the application logic as a client-side only centralized state Web application exploiting the unidirectional data flow pattern. This approach allows for a easy-to-serialize state (in the form of a JSON document) that can be pushed on a third party document oriented DB-as-a-Service and loaded back in the frontend reactive architecture, which transparently reload the state once its serialized version is passed in. The application itself is served by a CDN (Content Delivery Network) thus avoiding any need for web server. Offline routines rely on Function-as-a-Service platform as well as users management and collaboration features.

This architecture has been successfully employed by the authors in the *Metior* project [10], a tool to support selective deconstruction of buildings in the pursuit of a "zero waste" model.

Acknowledgments

Authors would like to thank GEOWEB S.p.A., a web service company owned by Sogei S.p.A. and CNGeGL Italian National Board of Quantity Surveyors, for supporting this work. Thanks are extended to Stefano Perrone for developing the models shown in the Figures 1, 2, 3 and 4.

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.

[10] E. Marino, F. Spini, D. Salvati, C. Vadalà, M. Vicentino, A. Paoluzzi, and A. Bottaro, "Modeling semantics for building deconstruction," in *Proceedings of the 12th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - GRAPP*, 2017, to appear.