A code generator for building front-end tier of REST-based rich client web applications

Nikola Luburić, Goran Savić, Gordana Milosavljević, Milan Segedinac, Jelena Slivka
University of Novi Sad, Faculty of Technical Sciences, Computing and Control Department
{nikola.luburic, savicg , grist, milansegedinac, slivkaje}@uns.ac.rs

***Abstract* – The paper presents a code generator for creating fully functioning front end web application, as well as a JSON-based DSL with which to write models which the generator uses as input. The DSL is used to describe both the data model and the UI layout and the generated application is written in AngularJS, following the current best practices. Our goal was to produce a code generator which is simple to create, use and update, so as to easily adapt to the climate of technologies which are prone to frequent updates. Our code generator and DSL are simple to learn, offer quick creation of modern, feature rich, web applications with customizable UI, written in the currently most popular technology for this domain. We evaluate our solution by generating two applications from different domains. We show that the generated applications require minor code changes in order to adapt to the desired functionality.**

INTRODUCTION

In recent years classical desktop applications have been replaced by internet-based applications where a server provides core functionalities that are accessed from client applications. In software engineering the terms "front end" and "back end" are distinctions which refer to separation of concerns between a presentation layer and a data access layer respectively. A recent trend in the implementation of internet-based applications is to separate the logic in two independent applications, a back end application which runs on a remote server and a front end application which runs on the browser.

Communication between client applications and the server is mostly done over HTTP, based on the REST software architecture [1]. REST-based services provide clients with a uniform access to system resources, where a *resource* represents data or functionalities, where each resource is identified by its uniform resource identifier (URI). A client interacts with such services through a set of operations with predefined semantics. REST-based services typically support CRUD operations which, in the context of internet-based applications, map to HTTP verbs.

In the recent years there has been an expansion of new technologies for developing front end applications and from year to year the growth of available frameworks and libraries is exponential [2-4]. Likewise, the vast majority of those technologies are volatile and tend to differ significantly from version to version. While few frameworks and libraries have proven to be more than hype, like AngularJS and Bootstrap, even those are prone to upgrades that break legacy software (which is rarely more than a year old). One thing to note is that while the backend technologies are constantly improving, the improvements made in this field revolve around performance or making the developer's lives easier, while the features, visual appeal and ease of use of front end applications are what bring users in and influence profit [5]. This, in turn, means that there is more to be gained from updating the user interface, than the underlying server application.

Regardless of the technology being used, most information systems contain a standard set of features and functionalities. Features like CRUD (*create*, *read*, *update*, *delete*) forms or authentification are part of most applications, which is why it is possible to create tools which will generate these features automatically. One thing to keep in mind is that the tool would need to change as frequently as the underlying technology, and when talking about technologies which are prone to change and frequent updates, both the generator and its input need to be simple enough in order to be useful.

This paper presents a code generator for generating the front end tier of rich client web applications that rely on REST services as a back end technology. Our generator uses a simple DSL based on JSON as input, which is presented as well. The DSL describes a business application and the generator uses that description to generate an application. We use this code as a starting point of the implementation, giving a head start to the development process. The generated client application is written in AngularJS, using the current best practices [6]. In order to evaluate our solution we have performed two case studies. Using our code generator we have automatically generate implementations for two applications from different domains: a registry of cultural entities and a web shop for a local board game store. We show that the generated applications need minimal modifications in order to be customized according to the specific requirements. While the DSL is technology agnostic, the AngularJS framework [7] was chosen for the generator as it is currently the most popular framework for developing front end web applications. The reason behind its popularity lies in its ability to extend HTML through directives, while offering dependency injection in JavaScript which reduces the number of lines of code written. It also provides two way data binding between the view (HTML) and model (JavaScript) and offers many more features that increase the quality of web applications while minimizing the amount of written code.

The paper is organized as follows. Work related to this paper is presented in the next section. The section „Input DSL" presents the DSL we use to create our input model for the generator. The section „Code Generator" describes our code generator. The section after that, titled „Case

Study“, presents two applications which were created using our generator as a starting point and shows the amount of code that was generated which required no modification, as well as the amount of code which required some modification or which had to be manually writen.

RELATED WORK

Before developing our solution we considered both the current research in the scientific community and the current industry standards (the popular open-source tools).

In [8] Dejanović presents a complex DSL which is used for the generation of a full stack web application using the Django framework. The DSL covers many different scenarios in order to automatically generate as much code as possible, including defining corner case constraints, custom operations, etc. The resulting DSL is a complex language that requires time and effort to learn. Code generators based on this language are complex and require a lot of time to be implemented if every part of the language is to be covered, which means that such a solution can’t be used in a climate where every new project works with a different technology, or at least a significantly different version of the same technology. Paper [9] presents a mockup driven code generator, which is easier to use and requires less effort on the part of the developer, while also offering a significant amount of configuration as far as the user interface is concerned. However, the tool is also far too complex for the group of technologies being examined. The likely scenario is the long development of the code generator itself. With fast changing technologies this has a consequence in generation of already deprecated code. It should be noted that both [8] and [9] are aimed at generating enterprise business applications, which require mature and stable technologies. It should be noted that while both solutions take a MDE (model-driven engineering) approach, our generator focuses on creating a starting point for the implementation of an application.

With regard to code generators for front end web applications the Yeoman scaffolding tool [10] is a popular tool for this area. This tool provides developers with an improved tooling workflow, by automatically taking care of the many tasks that need to be done during project setup, like setting up initial dependencies, creating a suitable folder structure and generating the configuration for the project build tool. Most modern code generators use the Yeoman tool for the first step of building a front end application.

Some solutions that build on the Yeoman tool focus on expanding the scaffolding process, initially creating more folders and files based on some input. While no real business logic is generated, the files are formed with sensible defaults and best practices. The angular generator [11] by the Yeoman Team and Henkel’s generator [12] are the most popular solutions from this group of generators. While the use of these tools is easy and usually requires only strings as input, the resulting code isn’t runnable, as it’s mostly boilerplate code.

A second group of solutions that build on the Yeoman tool try to produce fully functioning applications based on some input, and this is where our generator and DSL fit in. The most popular tool in this area by far is JHipster [13]. Using the command line or XML as input, JHipster creates a fully functioning application (both back end and front end) written using Spring Boot and AngularJS. While JHipster does offer a lot of useful features (built-in profiling, logging, support for automatic deployment, etc.) and the back end application is well implemented, the disadvantage of this tool is the lack of a fully developed front end application. The generated front end application lack important features (GUI elements for many-to-one relationships and lacks any customization of the layout during code generation, which means that every generated application looks exactly the same.

Aforementioned solutions are either too simple and generate only boilerplate code and/or only take the data model into account when building the user interface and/or are too complex for building cutting edge front end web applications. When comparing our solution with solutions produced by the scientific community, we found that the DSLs and code generators were far too complex for our problem domain of generating applications in technologies prone to frequent updates. Using our DSL users can describe not only the required model properties but also the layout which the generator uses to create a custom, well-designed GUI. In the next chapters we present our DSL and the code generator that uses it as input. Our goal here was to create a tool which solves the problems that the previously mentioned solutions have.

INPUT DSL

When constructing our DSL we aimed for simplicity. With that in mind we created a DSL which uses JSON as the underlying format primarily because our assumption is that a front end developer must know JSON and therefore doesn’t need to spend time learning the syntax of the DSL.

Our DSL needs to meet the following requirements:

- It describes browser-based applications that receive/send data through the network from/to RESTful web services contained within the server-side application. It describes not only the entities that the application handles (data model), but also the user interface (layout, components, etc.)
- It is simple so that developers can learn it quickly and its associated code generators can be developed efficiently, as we are targeting an area of software engineering known for its many frameworks which change rapidly [2-4]
- There is no redundancy in the description, known as the DRY (*don’t repeat yourself*) principle
- It is extensible, so that domain specific UI components can be built and used in the generating process.

Since an instance of our DSL is actually a JSON object, we can describe the constraints of our DSL using the JSON Schema [14].

Our code generator creates two components, a page for viewing multiple entities of a given type (list view, fig. 3), which supports paging, filtering and sorting of the list, and a form for creating a new entity, or viewing and possibly editing an existing one (detail view, fig. 4). The generator takes a JSON document for each entity that we want to generate components for. The list and detail view are generated for each such entity, as well as the entire underlying infrastructure needed for the aforementioned views to work and retrieve data from the server.

We take several factors into account when describing our views – will the table have standard pagination or will it use infinite scroll, will our complex forms be segmented by collapsible sub-forms, using a wizard-like UI with next and finish or have no segmentation, etc. Listing 1 shows the part of the schema that describes the entity object. Note that only the identifier (`id`) and the `groups` element is required, while the rest are either generated using the identifier (`label`, `plural`), or have predefined values (`pagination`, `groupLayout`).

```
"title": "Entity",
"type": "object",
"required": ["id", "groups"],
"properties": {
  "id": { "type": "string" },
  "label": { "type": "string" },
  "plural": { "type": "string" },
  "pagination": {
   "enum": ["default", "infiniteScroll"]
  },
  "groupLayout": {
   "enum": ["collapsible", "wizard",
            "none"]
  },
  "groups": {
   "$ref": "#/definitions/groups"
  }}
```

Listing 1. JSON schema for entity object

The `groups` attribute is an object which contains an `id`, optionally a `label` and an array of `attributes` and optional `subgroups`. By grouping attributes and subgroups we can separate complex forms into smaller, more manageable sub-forms.

An item of the attributes array contains information regarding the identity of the item (`id`, optional `label`), attributes which describe the type of UI component (`type`, `ref`, `object` and `extension`), and information regarding the list view (`table`).

Listing 2 shows the part of the schema related to defining the type of UI component for the given attribute of the entity. None of the listed attributes are required and if the `type` is missing it will default to *string*. If the `type` of attribute is one of the last three listed (*ref, object, extension*), another attribute is needed (which has the same name as the value of the type attribute) which will further describe the UI component. In case the `type` is *ref* the `ref` object describes a relationship with another entity, as well as how to form the UI component (which is defined in the `presentation` attribute). In case our REST back end returns an object that contains an inner object, we use the `object` attribute along with setting the `type` to the previously defined *object* we need. Finally, we use `extension` attribute to define custom components, by supplying a simple string which the generator will process in its own way. As far as our generator is concerned, strings listed in the `extension` attribute are angular directives.

The **`table`** object, not presented in the listings, is related to the functionality and UI of the list view. Three flags are placed in this object, `show` which signifies whether the attribute should be displayed in the table, `search` and `sort` which enable/disable the search and sort functionality of the table for the given attribute.

```
"type": { "enum": [
  "string", "textArea", "number", "email",
  "date", "ref", "object", "extension"
]},
"ref": {
  "type": "object",
  "required": ["entity", "relation"],
  "properties": {
    "entity": { "type": "string" },
    "relation": {
      "enum": ["oneToMany", "manyToOne",
               "oneToOne"]
    },
    "independant": { "type": "boolean" },
    "presentation": {
      "enum": ["inline", "link", "none"]
    }
  }
},
"object": {
  "type": "object",
  "required": ["id", "attributes"],
  "properties": {
    "id": { "type": "string" },
    "attributes": {
      "$ref": "#/definitions/attributes"
    }
  }
},
"extension": { "type": "string" }
```

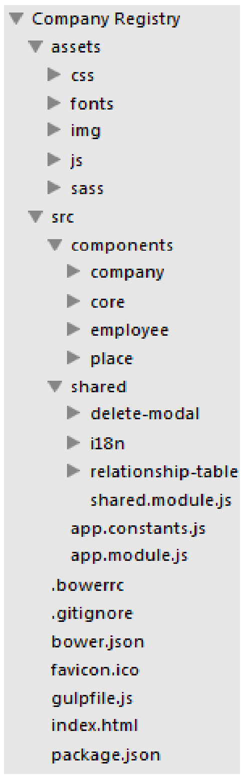
Listing 2. JSON Schema for attribute type definition

CODE GENERATOR

Our code generator uses instances of our JSON schema presented above to create components for a fully functional front end web application.

The code generator uses the Freemarker template engine. The generator uses the input model writen using our DSL and template files to produce the resulting application. The template files are closely related to the chosen technology, and our templates are created for the latest version of the popular AngularJS framework, along with the Bootstrap CSS library in order to provide a rich, responsive modern front end web application.

Apart form the code generator, a framework was developed which acts as the infrastructure for the generated code. Other than a few input strings (e.g. application name, remote server location) the framework doesn't require any additional information. The generated applications are written using the current best practices for project structure and angular coding styles, contains inbuilt support for internationalization and use visually appealing and intuitive UI components. If a new entity needs to be added into the system, one would only have to supply the generator with the appropriate JSON and copy the resulting folder into the components folder. Fig. 1 shows the folder structure of a generated application with three entities. The content of the `components` folder is what the code generator produces, while everything else is part of the framework.



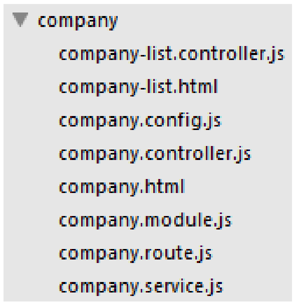
```
▼ Company Registry
  ▼ assets
    ▶ css
    ▶ fonts
    ▶ img
    ▶ js
    ▶ sass
  ▼ src
    ▼ components
      ▶ company
      ▶ core
      ▶ employee
      ▶ place
    ▼ shared
      ▶ delete-modal
      ▶ i18n
      ▶ relationship-table
        shared.module.js
      app.constants.js
      app.module.js
    .bowerrc
    .gitignore
    bower.json
    favicon.ico
    gulpfile.js
    index.html
    package.json
```

Figure 1. Generated application package structure

The `assets` folder contains resources which our application uses. This includes external JavaScript files (including angular and its plugins), styling sheets and fonts and images. The `src` folder contains the actual code of the application and it is separated into two subdirectories, the `components` folder and the `shared` folder. The `components` folder contains packages which represent various entities in our application and are, for the most part, independent modules which can be taken out of our application and placed into any other angular application where minimal work would be needed to adapt the module to work in the new context. The `core` package contains items that define the application layout, like the homepage, sidebar, header and footer HTML files and the underlying controllers. The `shared` folder contains directives, services and other components which are used throughout the application. This folder, along with the `core` package of the `components` folder, makes up the infrastructure of our generated application. The components located in `shared` are reusable pieces of code which can be used in other applications with virtually no adaptation required. This package includes support for internationalization, a directive for displaying one-to-many relationships and a modal dialogue which prevents accidental deletion. Finally the generator provides files for tracking bower and npm dependencies, which list dependencies of our application and development tools (gulp for managing the build process, karma and protractor for running unit and end-to-end tests) respectively, and a gulp file which contains a set of commonly used tasks.

The code generator creates specific UI elements for associations. The many-to-one relationship is represented using an autocomplete textbox, which offers a list of results that are filtered using the user input. The one-to-many relationship is displayed using an inline table.

Coming back to the generation process, the second phase of application generation takes all the JSON files written in accordance with our schema to produce modular, independent components, which include a list view page and detail view page described in the previous chapter, underlying angular controllers for both pages, a routing configuration file for application state transitions and a service which communicates to the REST endpoints on the remote server. Fig. 2 shows the content of a folder for one of our entities, where the previously described files are listed.



```
▼ company
    company-list.controller.js
    company-list.html
    company.config.js
    company.controller.js
    company.html
    company.module.js
    company.route.js
    company.service.js
```

Figure 2. Generated component based on the input DSL

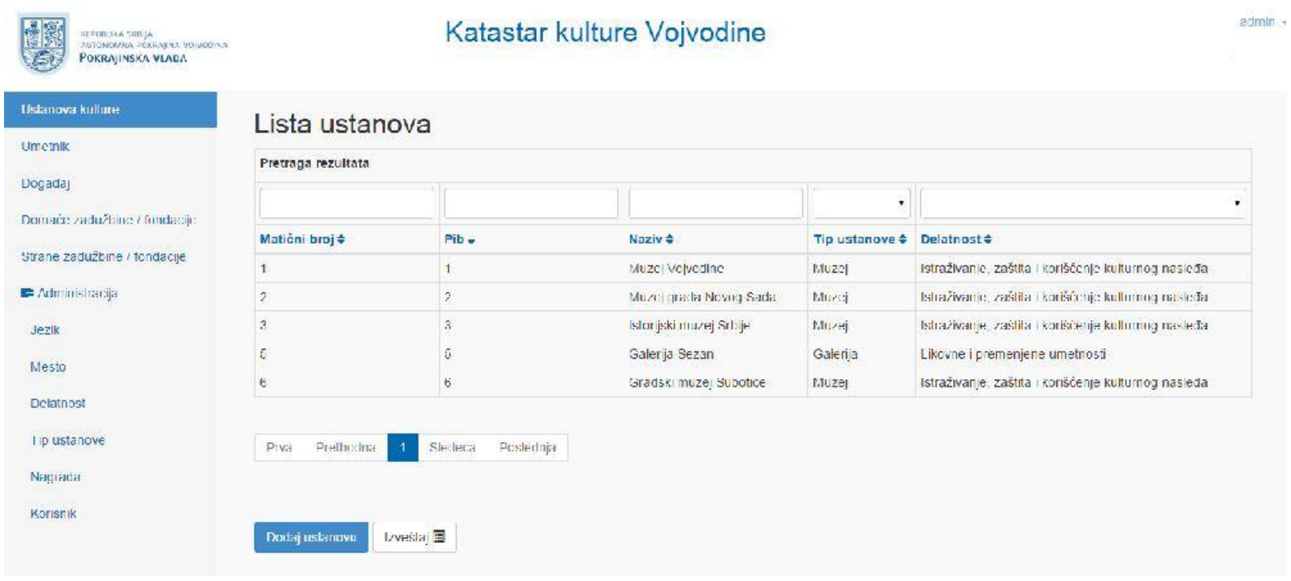
CASE STUDY

This section presents two applications, a registry of cultural entities and a web shop for a local board game store. The first system works with 56 entities, but doesn't require animations or other forms of highly interactive interface. The second system deals with a few entities, but requires a dynamic interface with a lot of animations. Both applications have subsystems which aren't covered by our code generator and have to be implemented manually.

The registry of cultural entities is an information system which records information about cultural institutions, artists, cultural events, domestic and foreign foundations and endowments in AP Vojvodina. Each one of these entities has over ten entities related to them, and some of these entities are shared between the primary five. When speaking in the context of a relational database, the data model consists of over seventy tables, which include tables for recording multilingual information, as well as tables for tracking changes of various entities of the system.

Fig. 3 shows the generated application, localized to Serbian, and its basic layout, where the header and sidebar are mostly static, while the workspace containing the list of institutions changes. The displayed list view is generated for the cultural institution entity. Since this entity has over thirty fields (counting related entities) only a small subset was chosen to be presented in the table. The table offers support for pagination, sorting and filtering based on the displayed attributes. Whenever we want to display multiple entities of a given type we use a table similar to the one displayed in the figure.

By clicking on an item in the table a form for viewing and editing the clicked item is displayed on the workspace, as shown on fig. 4. Most of the form shown in fig. 4 is created using the components the generator provides, like a date picker, an autocomplete textbox for many-to-one relationships and an inline table for one-to-many relationships. A custom UI component which represents a table for adding multilingual data was created as an angular directive (marked with the red square in fig. 4)



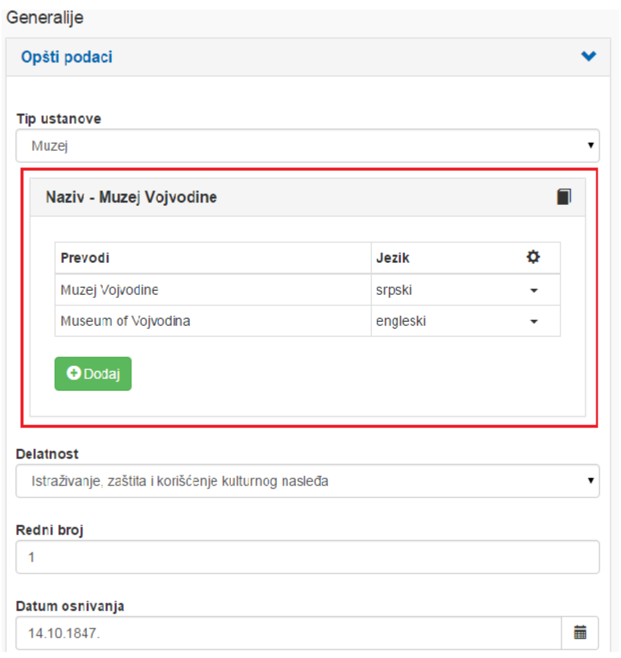
Lista ustanova

Pretraga rezultata

Matični broj	Pib	Naziv	Tip ustanove	Delatnost
1	1	Muzej Vojvodine	Muzej	Istraživanje, zaštita i korišćenje kulturnog nasleđa
2	2	Muzej grada Novog Sada	Muzej	Istraživanje, zaštita i korišćenje kulturnog nasleđa
3	3	Istorijski muzej Srbije	Muzej	Istraživanje, zaštita i korišćenje kulturnog nasleđa
5	5	Galerija Sezan	Galerija	Likovne i premenjene umetnosti
6	6	Gradski muzej Subotice	Muzej	Istraživanje, zaštita i korišćenje kulturnog nasleđa

Figure 3. Application layout and list view of cultural entities

and was listed (using the name of the directive) in the input DSL, under `type` *extension*. In this way we have used the extensibility feature of our DSL.



Tip ustanove: Muzej

Naziv - Muzej Vojvodine

Prevodi	Jezik
Muzej Vojvodine	srpski
Museum of Vojvodina	engleski

Delatnost: Istraživanje, zaštita i korišćenje kulturnog nasleđa

Redni broj: 1

Datum osnivanja: 14.10.1847.

Figure 4. Detail view of a cultural entity

The board game web shop worked with 11 entities but required more work on the UI and UX front. Fig. 5 shows the resulting application. The templates were slightly modified to create a more colourful UI. The detail view has a similar look to the previous application, only the attributes of the entity are displayed as labels and not as input controls and there are fewer fields.

During construction of the listed applications a portion of the code was generated and this code required very little or no modification. A portion of the code was generated but required significant modification, usually by using the generated code as a starting point.

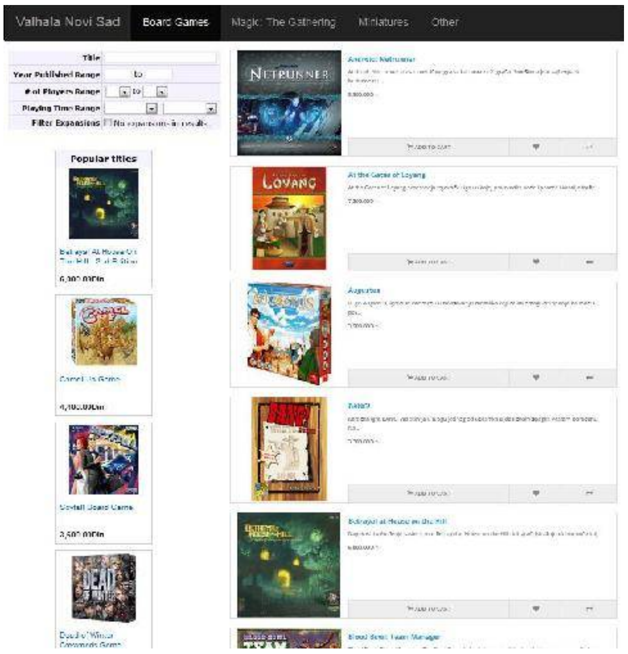


Figure 5. Application layout and list view of web shop

Finally, a portion of the code had to be manually writen so that the system would meet the needed requirements. The percent of generated JavaScript and HTML code that required no or significant modification, as well as the percentile of manualy written code for both applications can be found in table 1.

The front end application for the registry of cultural subjects had about 80% of the code generated which required no or very little modification. Another 10% of the code was generated but required some modification, and this included specific constrains on the forms and custom UI which was different from what our generator provided. The remaining 10% of the code had to be manually written, and this included the subsystem for user authentication, a service for contacting the server to initialize report generation using the WebSocket protocol [15], and a custom homepage.

TABLE I.
PERCENTILE OF GENERATED AND MANUALLY WRITTEN CODE

Application	Registry of cultural subjects	Board game web shop
Number of entities	56	11
Number of pages	19	13
Lines of JavaScript code	7303	2105
Lines of HTML code	5508	985
Generated JavaScript with no or little modification	75%	65%
Generated HTML with no or little modification	90%	65%
Generated JavaScript with significant modification	15%	10%
Generated HTML with significant modification	5%	5%
Manually written JavaScript	10%	25%

The board game web shop required more work on the UI and UX front. While AngularJS does have good support for animations, this wasn't a primary requirement of our code generator, which is the reason why almost no animation and dynamic interface behaviour was generated.

For the front end side of this system about 65% of the code was generated and required no or very little modification, while another 10% required a decent amount of change. The remaining code had to be manually written, and this included the subsystem for making purchases (a shopping cart) and upgrading the UI with animations and graphics.

CONCLUSION

The paper presents a code generator used for creating rich front end web applications, written using the popular AngularJS framework. The generated applications are written following the current best practices for project structure and angular coding styles, contain inbuilt support for internationalization and use visually appealing and intuitive UI components. As input, our code generator uses a model writen in our simple DSL, based on JSON, in order to be easy to learn and in order to avoid overly complicated code generators which take more time to develop than a new version of the technology used in the generated code. Our DSL supports description of both the data model and the user interface layout and components in a concise manner. The code generator uses instances of this DSL as simple JSON objects to construct fully functional applications build with AngularJS and the Bootstrap CSS library.

The DSL and the code generator have been evaluated by creating two applications from different domains. Compared to other similar solutions the generator was either more flexible, by allowing the developer to define both the data model and the layout of the application and/or was easier to use, avoiding corner case constraints and details in implementation and/or was more complete, by generating a full application rather than just project scaffolding.

Our current solution only takes into account the user interface layout and the data model from the REST endpoints. An important part of data driven applications are constraints on user input, and this is something that our DSL and code generator currently don't support. Furthermore, our DSL only takes into account the data model from the REST endpoints and makes the assumption that all applications follow the REST-full pattern for managing entities using a limited set of operations with predefined semantics. This could be a limitation for the practical use of our generator since many systems rely on REST-like web services which do not have a predefined set of methods for manipulating entities.

The future plans for the DSL and code generator include:

- Separating the current DSL into separate logical units, one for defining the data model and its constraints and one for describing the UI layout
- Extend the generator to support REST-like services
- Extend the code generator to support generation of different types of applications, like mobile and desktop and in different technologies

ACKNOWLEDGMENT

Results presented in this paper are part of the research conducted within the Grant No. III-44010, Ministry of Education, Science and Technological Development of the Republic of Serbia.

REFERENCES

[1] R. Fielding, R. Taylor (2002), Principled Design of the Modern Web Architecture, *ACM Transactions on Internet Technology* (*TOIT*) (New York: Association for Computing Machinery) 2 (2), pp. 115–150, ISSN: 1533-5399

[2] A. Gizas, S. Christodoulou, T. Papatheodorou (2012), Comparative Evaluation of Javascript Frameworks, *Proc. of the 21st International Conference on World Wide Web*, pp. 513-514

[3] D. Graziotin, P. Abrahamsson (2013), Making Sense Out of a Jungle of JavaScript Frameworks, *Product-Focused Software Process Improvement, 14th International Conference (PROFES)*, pp. 334-337, ISSN: 0302-9743

[4] A. Domański, J. Domańska, S. Chmiel (2014), JavaScript Frameworks and Ajax Applications, *Communications in Computer and Information Science (CCIS)* 431: 57-68, ISSN: 1865-0929

[5] C. Kuang, Why good design is finally a bottom line investment, http://www.fastcodesign.com/1670679/why-good-design-is-finally-a-bottom-line-investment, retrieved: 23.11.2015.

[6] J. Papa, Angular Style Guide, https://github.com/johnpapa /angular-styleguide, retrieved: 23.11.2015.

[7] N. Jain, P. Mangal, D. Mehta (2014), AngularJS: A Modern MVC Framework in JavaScript, *Journal of Global Research in Computer Science (JGRCS)* 5 (12): 17-23, ISSN: 2229-371X

[8] I. Dejanović, G. Milosavljević, B. Perišić, M. Tumbas (2010), A domain-specific language for defining static structure of database applications, *Computer Science and Information Systems* 7 (3), pp. 409-440, ISSN: 1820-0214

[9] G. Milosavljević, M. Filipović, V. Marsenić, D. Pejaković, I. Dejanović (2013), Kroki: A mockup-based tool for participatory development of business applications, *Intelligent Software Methodologies, Tools and Techniques (SoMeT), 2013 IEEE 12th Inter'l Conference on*, pp. 235-242, ISBN: 978-1-4799-0419-8

[10] Yeoman Team, Yeoman, http://yeoman.io/, retrieved: 23.11.2015.

[11] Yeoman Team, AngularJS generator, https://github.com/yeoman /generator-angular, retrieved: 23.11.2015.

[12] Tyler Henkel, AngularJS Full-Stack Generator, https://github.com /DaftMonk/generator-angular-fullstack, retrieved: 23.11.2015.

[13] JHipster, http://jhipster.github.io/, retrieved: 23.11.2015.

[14] Internet Engineering Task Force, JSON Schema, Internet Draft v4, http://json-schema.org/, retrieved: 25.11.2015.

[15] Internet Engineering Task Force (2011), The WebSocket Protocol, *RFC 6455*, https://tools.ietf.org/html/rfc6455, retrieved: 26.11.2015.