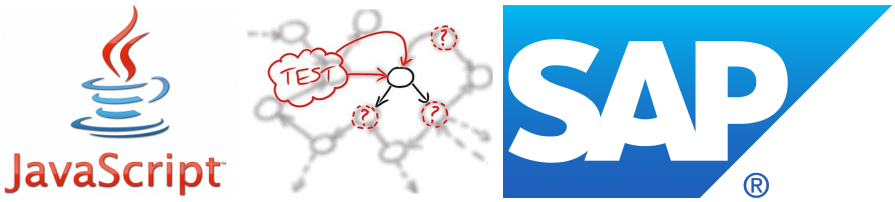
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Automatic Generation of Mock Data

Tomáš Bruckner*



Abstract
This work deals with creating and providing mocked data for applications that use REST interface to communicate between the client and server parts. From the various implementations of the REST interface, the work focuses only on OData standard, that is defined by Microsoft. The project itself is mainly for SAP company. Naturally, even the libraries that are used in the final solution are from SAP. Primaly JavaScript framework SAPUI5 is used. The merit of this work is a library that facilitates the development of the client side of web applications. It fully supports CRUD operations over OData calls. Compared to other libraries creating mocked data that always return the same static data, this one simulates the behavior of the real server. So, when DELETE method is called for a specific entity, the given entity is deleted. This functionality is enabled by the client-side database created directly in the web browser, which corresponds to the database on the server side. A similar library for OData protocol does not exist, so it is a unique solution. The solution is verified using prepared web application.

Keywords: SAP — REST — OData — mock data

Supplementary Material: https://github.com/SAP/openui5/

*xbruck02@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Testing is essential for all great projects. Testing is what decides if the project will live long and good life or if it will be that weird cousin from distant relatives that you do not want to talk about. That is why I have chosen this project. I worked on several projects that were using development methodology called *Edit and Pray*. I think it is self-explanatory.

In my project, I wanted to do something that has not been done before. I wanted to make a tool, which will automatically generate mock data for the web applications. The basic idea is to define an API, and the tool takes care of everything else. There are already a couple of tools that do something like this. So how is my tool different? It offers real CRUD operation. Other tools just fake responses to server side by responding with predefined static data. My tool can do it better. If you call POST operation under my fake server, the new entity is added to the database. In the following chapters, I will try to explain what is the problem and how it was solved in my tool.

2. Problem Definition

Every modern web application is using *Three-tier architecture*. That means that the application is divided into three independent modules. The database where all the data is stored. The server side that has the business logic of the application. The client side that renders what the user of the application see. At least that was the state of the game for a long time. In past ten years, everything changed. With the increasing computational strength of home computers and Inter-

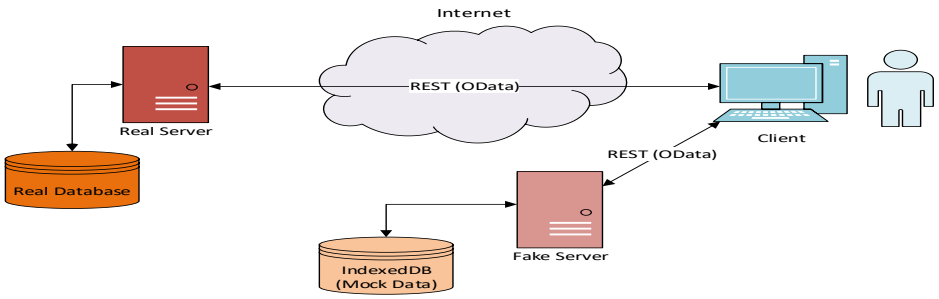


Figure 1. Schema of how a fake server works.

net speed, a lot of business logic has been moved to the client side. That's where JavaScript language shines. There are no other options today in web applications. Java servlets have never gained much popularity in the mainstream. Also Flash is being slowly deprecated in modern browsers, mainly because of security issues [1].

So now we established why JavaScript is the most important programming language right now. For server side, you can use any language you want, but on the client side, there is only JavaScript. So how does the development of *Three-tier architecture* looks? You defined API between client and server. It is implemented using *REST* architecture [2] that is part of the bigger set called *Microservices* that is part of an even bigger set called *Service Oriented Architecture (SOA)*. After the API is defined, the development can start. So where is the possible complication? You want to develop the client side independently from the server side. You don't want to wait for server side team to develop things you need.

So how to avoid this situation? You know the API so you can mock the responses. You define what static data should be returned on each request in API. But there is the problem. For example, we define that service on URL */entities* should return a collection consisted of entity X and entity Y for GET HTTP method. Everything is good so far. But then we add new entity Z to the collection with POST request to */entities*. The standard fake server returns OK that the entity was added to the collection without any problem. But that is only because we defined that the fake server should return OK on every POST request to that service. When we send GET request again, we only get X and Y again. That's because the fake server returns only static data. But why is that? Because if it was able to add the entity to the collection, it would already be the real server with the real database! And we don't want to spend time developing a fake server for each project. We want to specify API, and everything else should be automatic.

3. Existing solutions

My work is supported by SAP company. They are using *OData* protocol as their implementation of REST architecture. This protocol has been developed by Microsoft as an attempt to make it a standard for OData. Microsoft felt that reinventing the wheel every time for every project is not necessary. It is not needed to define new API for each project, when it can be defined by OData. *OData* has been standardized by OASIS in 2014 [3]. The advantage of the application using *OData* protocol is that the API is defined in *Metadata.xml* file that is fetched before the application starts. The metadata is not exactly API, but more specific description of the data model. The API is generated based on this model.

What can fake servers that generates mock data do today? There are a couple of existing solutions, but all of them work as described above. They serve static data. One of the most popular solution is *Apiary* [4]. Apiary allows users to define their API and responses for specific requests. It offers hosting the fake server with specified mock data on Apiary's server or you can host it on your own server. Another popular fake server is *MockServer* [5]. It allows verification of the defined requests, defining responses for the requests, execution of a callback or forwarding a request. No fake server can dynamically add, delete and modify entities. But it can greatly improve the way web applications are developed. It can improve testing of many use cases that are depending on dynamic data. We can be more certain that when we integrate the client side and the server side together, it runs more smoothly.

What I developed is a unique solution. As a developer, you get automatically this fake server serving mock data without any effort. The only precondition is that your application must run on *OData*. Let us explore in the next chapter how is the fake server implemented.

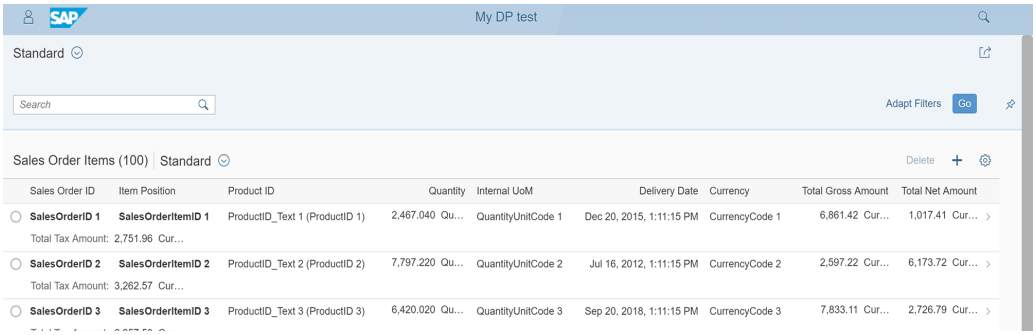


Figure 2. Demonstration of generated data.

4. Design and Implementation

The basic schema of the fake server is described in Figure 1.

The fake server is implemented as a part of *SAPUI5 JavaScript* library [6]. But how does it holds all the entities? In JavaScript, there are several ways to do it. You could use *SessionStorage/LocalStorage*, which is a *key-value* data store that enables only string values. That was quite inconvenient, and it would mean the server would have to serialize and deserialize everything. Searching would work extremely slow.

Next possible database type to consider was *WebSQL*. It is basically SQL database inside the browser. It is based on *SQLite* standard. But still, there are a couple of problems with this. Primaly Microsoft and Mozilla had objections about the *WebSQL* database so it is not implemented inside IE, Edge or Firefox [7]. The second inconvenience is that the SQL is not good fit for JavaScript language. You work with objects all the time, and some kind of *NoSQL* database would work so much better. That is why *IndexedDB* has been created. It is *NoSQL* database supported by all browsers. It supports database indexes and cursors.

Another core part of the fake server is catching of all *AJAX* requests so that they are not sent to the real server. This is done using the *SinonJS* library, that is already part of *SAPUI5*. It enables to start an instance of Sinon server that redefines *XMLHttpRequest* object that is used by JavaScript to send *AJAX* calls to the server. *SinonJS* allows you to specify what data should be returned for any specified request.

So the basic workflow of the fake server follow pattern. It fetches *Metadata.xml* file that defines the model and the API. Parses the file and creates the corresponding model in *IndexedDB*. It adds a few basic entities to the database. It adds rules using regular expressions to define what requests should be caught and defines callback functions that handle everything related to the database. Fetching, modifying, deleting and adding data to the database. The regular expressions are generated based on the model in *Metadata.xml*.

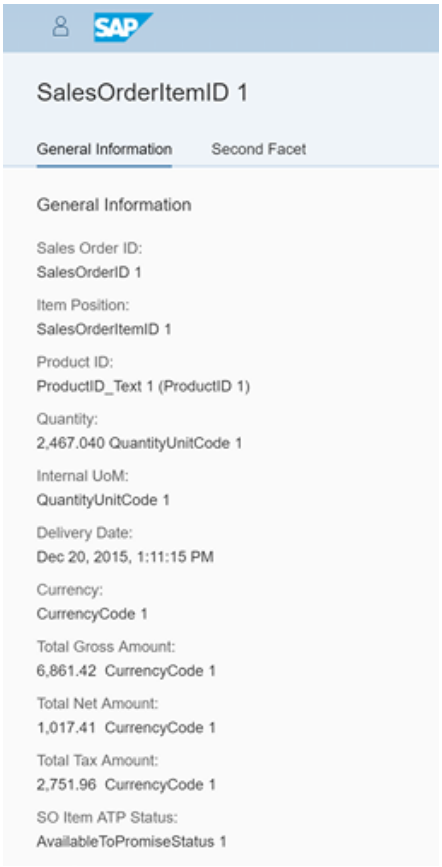


Figure 3. Demonstration of generated data.

For a validation of the solution, I have developed a few simple web applications that validate the implementation of the solution. They are based on Fiori Elements [8]. One web application is shown in Figure 2. It is a table based web application that has the option to add and delete new entities. In Figure 3, we can see the detail of the item. A user is redirected to this page when the user clicks on a table line. In detail, use has an option to see the full detail of the item or edit the item.

Based on the type defined in *Metadata.xml*, the fake server generated the corresponding data. For dates, emails and complex types, fake server can make a more specific mock data than for strings. For example for emails it generates the data in the format

email{*id*}*@email.com*. For ordinary strings, fake server concatenates name of the property with *_Text* {*id*}. It can be seen in Figure 2 in column *Product ID*. Fake server concatenates column *Product ID* with *_Text* and unique number.

The *Metadata.xml* file should contain *EntitySet* and *EntityType* entities that defines the API. The advantage is, that to use *OData*, the application already contains this file. The documentation of the *OData* is quite complex. You can find more information about the format in the documentation [3].

After the real server is developed, it is easy to switch the client side to use the real data. Easiest way would be to have variable that represents the environment of the code. Based on this variable, there could be a condition statement that either starts the server or does not do anything.

5. Conclusion

This paper deals with an advanced fake server that behaves almost like a real server. This makes the independent development of the client and server side much easier. What it lacks from the real server is its business logic, it just supports CRUD operations on the entities. To best of my knowledge, it is a completely unique solution that is not available anywhere else. Or at least there is no other publicly available solution with same functionality as my fake server. This library is a part of *SAPUI5* which is also an open source project that can be accessed on *GitHub*.

The work on this project will continue in the open source variation. *OData* is evolving standard and SAP adds its customization to the standard. Also, the tool supporting *OData* version 2 right now because this is the version SAP is using. In the future, there are plans in SAP to switch to version 4. If that happens, the fake server will have be made compatible with the version.

Acknowledgements

I would like to thank my supervisor Ing. Jan Pluskal for his help.

References

[1] Chrome to Deprecate Flash in Favor of HTML5. Security Week, 2016. [Online; visited 19/04/2017] http://www.securityweek.com/chrome-deprecate-flash-favor-html5.

[2] Leonard Richardson and Michael Amundsen. *RESTful Web APIs*. O'Reilly, 2013.

[3] OASIS. Open Data Protocol (OData) TC, 2016. [Online; visited 10/01/2017] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata.

[4] Dave Goldberg. The role of the api designer. Apiary Blog, 2015. [Online; navštíveno 30.12.2016] https://blog.apiary.io/2015/05/03/The-Role-of-the-API-Designer.

[5] James Bloom. Mockserver. MockServer, 2015. [Online; navštíveno 30.12.2016] http://www.mock-server.com/.

[6] SAPUI5: UI Development Toolkit for HTML5. SAP SE, 2016. [Online; visited 30/12/2016] https://sapui5.hana.ondemand.com/#docs/guide/95d113be50ae40d5b0b562b84d715227.html.

[7] Arun Ranganathan. Beyond HTML5: Database APIs and the Road to IndexedDB. Mozilla Hacks – the Web developer blog, 2010. [Online; visited 07/01/2017] https://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/.

[8] Introduction to SAP Fiori Elements. SAP SE, 2016. [Online; visited 31/12/2016] https://experience.sap.com/fiori-design-web/smart-templates/.