State-of-the-art Web Applications using Microservices and Linked Data

Aad Versteden and Erika Pauwels

TenForce, Havenkant 38, 3000 Leuven, Belgium
{aad.versteden,erika.pauwels}@tenforce.com

Abstract. This paper describes the current state of `mu.semte.ch`, a platform for building state-of-the-art web applications fuelled by Linked Data aware microservices. The platform assumes a mashup-like construction of single page web applications which consume various services. In order to reuse tooling built in the community, Linked Data is not pushed to the frontend.

Keywords: microservices, semantic model, REST API, Linked Data, RDF, SPARQL, triple store

1 Introduction

Considering the construction of modern web applications in the industry, single page apps have become king. Frameworks like EmberJS[1], Angular.js[2] and React[3] have taken the industry by storm. These frameworks make it easy to mash up various web services into a unified application. But how do these web services interact? Can we use Linked Data as a solution to connect various web services? `mu.semte.ch` offers a pragmatic answer to these questions with the easy construction, sharing and consumption of microservices.

The `mu.semte.ch` platform[4] does not start from the assumption that Linked Data has value, but uses Linked Data as an ally to solve the harder problems in a microservices based architecture. Under the assumption that microservices can provide an answer to the needs of a modern single page application, how would these microservices interact? If they need to share information, do they need to depend on each other? Microservice dependencies are one of the harder problems to crack in microservices based architectures[5], but Linked Data can offer an answer. By sharing a joint model and database, the services all depend on the same data layer, rather than on the existing services. This means a service can go down, or scale up, without affecting other services. We could share information on a web scale, without services being dependent on each other's existence, by sharing only the data layer.

We are using this platform in real-world scenarios to build web applications and share code across projects. Simplicity is key in order to minimize the cost of training, application development and system setup. The semantic model is not pushed to the frontend, in order not to retrain developers and in order to

make full use of existing tooling. Since our initial description of the platform, we have tested and adapted the platform to these real-world use cases. The platform has evolved from an initial idea to our prime platform for quickly developing practical applications on top of Linked Data. Core microservices have been constructed which can handle the bulk of traditional data retrieval and creation needs. Microservice templates have been built indicating how new languages can be supported. And a new way has been devised to easily share code both in the frontend as well as in the backend.

In this paper we describe the current state of the platform. We start with a short overview of the current platform, and why it works, and continue with state-of-the art ways of constructing, sharing, and displaying content. Finally, we describe some related work and the lessons we have learned so far. Throughout the paper we will use a voter app as example to clarify concepts and provide code examples. The voter app allows users to create topics which other users can upvote. A user can upvote a topic only once and must be able to withdraw his vote. An implementation of this application is available on GitHub[6].

2 Platform architecture

The `mu.semte.ch` platform enables extensive code-reuse for modern web-based applications. As single page applications stay alive in the frontend for extensive periods, even beyond network disconnects and page closes, the system can be considered to be a true distributed system. A clear split is made between the semantically enabled backend and the visualisation-centered frontend, as shown in Figure 1.

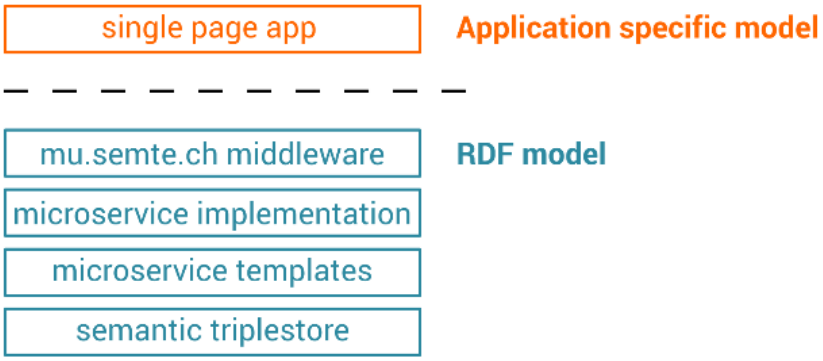

Fig. 1. `mu.semte.ch` platform stack

The frontend considers current state-of-the art solutions for building single page applications. The backend considers very simple isolated microservices. Aside from a limited set of core microservices, each microservice offers an answer

to a specific user problem without depending on other microservices. For example, the voter app will require microservices to register users, handle login, manage topics and vote on a topic. The glue between the backend and the frontend is the RESTful JSONAPI[7] format, which is becoming the de facto standard for communication in the JavaScript community. The general approach of the `mu.semte.ch` framework is described more in depth in [8].

2.1 Single page application

The single page application is powered by EmberJS[1] in our architecture. This single page application framework heavily bets on tooling and follows the mantra of sensible defaults and convention over configuration. For example, since the v1.13 release, it follows the JSONAPI standard[9]. EmberJS makes it also possible to easily share common code between applications in the form of addons. The `mu.semte.ch` framework currently offers addons for a login, registration and authorization management service[12].

For example, if we want to add a login screen to the voter app, the addon allows us to download, include and compile code for a login component by running

```
> ember install ember-mu-login
```

From there on, you can include `mu-login` in a frontend template and all logic for handling logins will be taken care of by the addon. This heavy focus on simplicity and code reuse through solid tooling is apparent in every layer of `mu.semte.ch`.

2.2 Middleware: mu-identifier and mu-dispatcher

Microservices should be as simple as possible, whilst still offering easy reuse. Two types of support are given for this case. The first level of support is in terms of the middleware layer described here. The second is by offering templates which offer clear guidelines on how to implement a microservice. Both are shown in Figure 1. In this section we describe the middleware.

The middleware serves two purposes: identification of a client, and routing a request to the correct microservice. The `mu-identifier` component[10] identifies the client by setting a secure cookie on the client. The cookie contains a tamper-proof URI identifying the specific client. This cookie is read by the `mu-identifier` which parses the cookie and sets the `MU-SESSION-ID` HTTP header on the request to indicate the session URI of the current client. All information connected to this client, like which user has logged in on this client, will be stored in the triplestore. The identifier therefore does not need to talk to the triplestore. The augmented request is forwarded to the `mu-dispatcher` component[11].

The `mu-dispatcher` receives the request and forwards it to the responsible microservice. This configurable behaviour makes it simple to host multiple microservices whilst avoiding collisions on the URLs used by each microservice.

With this approach, a payment service and a user-registration service can both offer a `/submit` route without colliding with each other. In our voting app the dispatcher could be configured to dispatch all requests starting with `/sessions` to the login service. Requests starting with `/vote` are dispatched to the plus-one microservice. Similar dispatching rules can be configured for registration and topic management. The complete configuration looks as follows:

```
match "/sessions/*path" do
  Proxy.forward conn, path, "http://login/sessions/"
end

match "/vote/*path" do
  Proxy.forward conn, path, "http://plusOne/"
end

match "/accounts/*path" do
  Proxy.forward conn, path, "http://registration/accounts/"
end

match "/topics/*path" do
  Proxy.forward conn, path, "http://resource/topics/"
end
```

2.3 Templates and microservices

The microservices maintain the business logic and manipulate the state of the application. They should be easy to understand because they have a limited scope. Developers receive a lot of freedom in the languages and technologies they prefer. The only constraints a microservice must adhere to are communication over HTTP, (meta-)information storage in a triple store and packaging in a Docker container[13].

Developing a microservice requires some initial work that needs to be done over again for each microservice. For example setting up a web server to consume HTTP requests and connecting to the triplestore to execute SPARQL queries. These functions are not core to the microservice and can easily be reused across microservices. In the `mu.semte.ch` platform they are therefore supported by templates. Each language which is supported on the platform has a template for building microservices. Currently templates for Ruby, Javascript and Python are available on GitHub[12]. Based on the development strategy of the target language, the template offers support for idiomatic ways of development (i.e. live code reloading), the inclusion and downloading of dependencies (i.e. installation of Ruby gems) and the building of new Docker images. The templating structure is also kept to a bare minimum. In order to create a new microservice which returns the votes for a particular item, the following code would suffice:

```
== Dockerfile ==
```

```
FROM semtech/mu-ruby-template:1.2.0-ruby2.1
MAINTAINER Aad Versteden <madnificent@gmail.com>

== web.rb ==
get '/:id/count' do
  resp = query "
   PREFIX votes: <http://mu.semte.ch/vocabularies/ext/votes/>
   PREFIX mu: <http://mu.semte.ch/vocabularies/core/>
   SELECT (COUNT(?user) AS ?votes)
   WHERE { GRAPH <#{settings.graph}> {
       ?user votes:plusOne/mu:uuid \"#{sparql_string params['id']}\"
   } } "

  status 200
  { data: { attributes: { votes: resp.first["votes"].to_i } } }.to_json
end
```

2.4 Triplestore: sharing knowledge between microservices

Microservices simplify application complexity as the scope of each service is limited. Each service offers very specific functionality by querying and altering a specific type of information. This information may, however, overlap between services. The `mu.semte.ch` platform leverages Linked Data to tackle the knowledge-overlap.

An example clarifies our case in point. Take user login and registration. A registration service could offer support for registering users, whereas a login service could offer support for users to login. The login service needs to know which users have registered. A joint data layer allows the services to operate independently of each other. If microservices are truly minimalistic, then they will invariably have an information overlap.

How do you share information between these microservices and how do you keep them in sync? Linked Data and semantic technologies offer a solution. Graph databases, like triple stores, can contain flexible models that can be queried and manipulated by all microservices. By using well-known and standardized ontologies to store the information we can ensure the services are talking about the same content in the same way.

A simple example expressing the prototypical case of a logged in user is presented in Figure 2. The grey zone of the picture is generated by the registration service. The blue content is generated when the login service identifies the user. It is clear that the registration service and the login service work in tandem to fulfill a complete login experience for the user. If both services speak the same language and have a similar understanding of what a login service entails, their implementation can evolve separately.

The services depend on ontologies for expressing the portion of the world our application is interested in. If the ontologies are chosen wisely, the application itself can evolve naturally. This also makes almost every microservice user-facing.

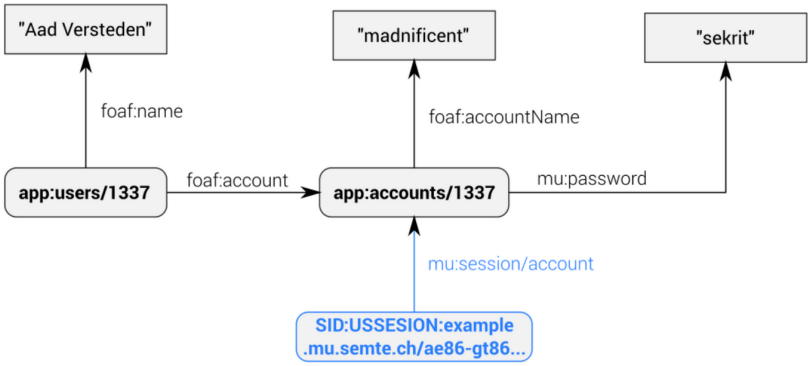

Fig. 2. User login example

Their API is presented -almost directly- to the frontend. This vastly simplifies the mental model of the application and makes future scaling easier.

The store uses SPARQL 1.1, including update semantics, offering an easy to understand and well documented technology for information storage. Where necessary, links to external systems can be made. For example to store a file uploaded by the user.

3 Industry solutions to a network of services

Microservices and single page applications have grown in popularity in recent years. Much development has occurred on both of these fronts. On the microservices front, Docker has paved the path in terms of deployment and maintenance. On the single page apps front multiple frameworks have stood up. More importantly, standards for communicating with these single page applications are on the rise with the standardization of JSONAPI. The state-of-the-art tools which we have seen in use in the industry are described in this section, they contain the context in which this platform operates.

3.1 Docker

The direction the Docker community is moving makes it trivial to host microservices. Docker allows Linux hosts to run extremely lightweight Virtual Machines. A virtual machine can contain all necessary libraries, except for kernel modules. The complex task of setting up microservices and keeping them running, has become trivial. The `mu.semte.ch` platform benefits from this by using the Docker ecosystem for sharing and maintaining microservices. Docker Hub[14] allows services to be shared easily and freely. This makes downloading and setting up microservices trivial.

3.2 Docker Compose

Docker Compose[15] provides a way to describe an ecosystem consisting of multiple microservices together with their configuration and connections in a single YAML file. The voter app could be described as follows:

```
version: '2'
services:
  identifier:
    image: semtech/mu-identifier:1.0.0
    ports:
      - 80:80
  dispatcher:
    image: semtech/mu-dispatcher:1.0.1
    volumes:
      - ./config/dispatcher:/config
  plusOne:
    image: madnificent/plus-one-service
  database:
    image: tenforce/virtuoso:1.0.0-virtuoso7.2.0
    environment:
      SPARQL_UPDATE: "true"
      DEFAULT_GRAPH: "http://mu.semte.ch/application"
    ports:
      - "8890:8890"
    volumes:
      - ./data/db:/data
  resource:
    image: semtech/mu-cl-resources:1.10.1
    volumes:
      - ./config/resources:/config
  registration:
    image: semtech/mu-registration-service:2.4.0
    environment:
      MU_APPLICATION_SALT: mysupersecretsaltchangeme
  login:
    image: semtech/mu-login-service:2.5.0
    environment:
      MU_APPLICATION_SALT: mysupersecretsaltchangeme
```

Mu-project[16] offers a good start to bootstrap such a new ecosystem in the `mu.semte.ch` framework. Docker Swarm[17] on its turn offers multihost deployment in a transparent way. It seamlessly integrates with Docker Compose allowing the user to deploy his ecosystem of microservices on a cluster with a single command.

As an example, downloading the necessary set of libraries, starting all microservices, and connecting them to each other, can be done with the following commands:

```
git clone https://github.com/madnificent/voter.git;
cd voter;
docker-compose up
```

3.3 JSONAPI

With the rise of single page applications, services offered on the web stopped hosting HTML pages. New services offer a JSON view instead. The proliferation of JSON-based APIs that resulted from this has been met with the community-based JSONAPI standard. This standard describes in which way resources should be shared across the web. In order to make applications easy to build, we have embraced this community standard. It minimizes discussion about the JSON responses which should be implemented. Removing this discussion turns the `mu.semte.ch` platform into an efficient tool for making a team cooperate.

The JSONAPI model is more ambiguous than a Linked Data model. Although the model can be well-defined within an application, sharing information across applications is not necessarily supported as semantics may differ per service. Using JSON-LD resolves this ambiguity. The type of the resource and its properties are well-defined and can be shared with any application. We sacrifice this general expressivity in order to match well with current best practices. These best practices are sometimes assumed by external libraries. Our choice thus makes it easier to reuse libraries built by the community and commonly used in the frontend.

Next to JSONAPI a number of other hypermedia formats have been defined such as HAL, Collection+JSON, Siren and Mason. These standards specify a serialization format, but do not specify the interaction (e.g. how documents can be created, updated and deleted). JSONAPI does describe those operations leaving less room for discussion.

4 Alternative approaches

Our approach is an example of a three-tier architecture[18]. A paradigm which is well-known and natural for developers. By representing the data in a semantically correct way, it becomes easy to represent the middle tier as a set of microservices. Our approach makes these microservices easy to download, install, and run, by using Docker. The approach also makes the microservices easy to run, by providing a general pattern to be used by the middle tier in most scenarios. Namely, the use of a webservice which yields JSONAPI in its response. The approach is therefore novel in the tooling used making it more practical in a modern world than previous approaches[5].

Although comments have been made that web applications often do not follow the idea of a three-tier architecture[19], this is not the case with microservices and single page web applications. The services can be consumed by other (single page) applications, whilst the single page applications can easily switch to services offered by external providers. The frontend cannot access the data layer directly at all. We do face the claim that a DBMS does not count as a tier, but feel that a lightweight semantic model of the domain model accessible through SPARQL may be sufficient. We need more experience to validate whether reasoning in the database is an absolute necessity or not. A store with inferences seems to address the concern that a DBMS does not count as a data tier.

Other frameworks have been built on top of linked data. Graphity[20] is one of them. We differ however in two major ways. The first is that we assume the processing of requests to be written as microservices, written in common programming languages, rather than trying to express everything in RDF. We consciously use RDF only to describe the domain model. This limits the paradigm shift for developers and allows the use of commonly known tooling. Secondly, we assume that the application serving the client will be riddled with custom requests. Hence we use an API to communicate with the microservices, and build a fully custom interactive UX. Although components can be reused, it seems that tiny changes are often necessary requiring flexibility at the frontend.

5 Lessons learned

Our experience with this platform so far has been very positive. Aside from the initial discovery of the strong and weak points of this architecture, it has shown its use in various projects covering various situations and domains. It is comparatively easy to get people started with the necessary technologies. Code reuse has shown to be plain and simple in real-world cases. We have also seen that it is easy to get new technologies included, though that does give rise to some questions.

We have used `mu.semte.ch` in quite a few projects since we have paved the first roads past year. First and foremost, quite a few of the extensions to `mu-cl-resources` microservice have been done to match the needs of the Your Data Stories project[21]. These changes have made this specific microservice a lot more flexible. Some of these changes have had a positive impact on internal projects supporting ESCO. The ESCO Mapping Pilot has been a pilot case using `mu.semte.ch` as a basis for ESCO software. Our evaluation was very positive and the subsequent ESCO Mapping Platform and ESCO Translation Platform have both used `mu.semte.ch` in an idiomatic way. Furthermore, the platform has been used to support webcat, a demo frontend for the DCAT standard. The platform has also shown its use in the development of the BDE-pipeline-app, an application for supporting the postponed launching of computations in the pipelines of the Big Data Europe project[22]. Last, but not least, a lot of lessons were learned whilst constructing MusicCRM, a platform for supporting the administration of music associations. The `mu.semte.ch` platform was contrived

whilst starting the development of MusicCRM, hence this application has gone through the complete evolution of the platform. It has been our drive to make upgrades smooth and simple. With the projects we have undertaken so far, we notice that developers which know the platform naturally shift towards using it for new applications. We suppose that the platform strikes a good combination of fun and efficiency.

It has been fairly easy to get new developers to develop new microservices. Experienced developers need little information to understand the freedom and constraints they receive in the backend, rookie developers get started easily because they can start in a (near-)greenfield setting. Our dissemination towards new users has mostly been handled orally. We are actively working on written documentation to get developers started on the platform. Some experienced developers have a natural tendency to stretch the constraints of the platform, which may pose a risk when sharing microservices. We need to see how the ecosystem evolves when direct human contact is not an option.

Introducing new developers to the frontend is straightforward. The frontend behaves as an idiomatic EmberJS application. It has been easy to guide people in the application as there are good books and guides around for this framework. Our extensive focus on tooling has also made the builds of this framework reproducible, splitting off another risk. Getting experienced EmberJS developers started, or letting them teach others is simple due to the many conventions in the framework. Developers not experienced with single page applications seem to need some time to grok the paradigm shift.

We have seen more code-reuse than we have seen with more traditional stacks we have used internally. We accredit this to the fact that a microservice, once written, needs no adaptations in order to be used within another application. Much unlike other approaches, the inclusion cannot easily break existing code due to mismatching dependencies. As both frontend and backend components can be shared, there is much drive for developers to develop components which can be reused early on. Due to this code-reuse we are looking at more generic building blocks which may be used in many applications (i.e. a component for paginating and sorting data tables). There is still low hanging fruit in this domain.

There is a lot of flexibility in the implementation-language and strategy for the microservices, which has its downsides. A common stack using the `mu.semte.ch` platform can easily use four programming languages. This is frightening to some developers and may hinder company-wide adoption. It may be hard to find developers to maintain all of these languages. We have noticed that most microservices we have developed can be rewritten in under ten days of development which alleviates this issue. When used carefully we have also noticed this to be of great benefit. For the Big Data Europe, the REST interface for deploying pipelines was written in Python so we could easily access the Docker Compose libraries. As we can choose the language which best suites the application to write, we can sometimes greatly limit development time.

So far, the platform has shown to be maintainable. Since the start, Music-CRM has been using this platform in its core. The application's development can evolve together with `mu.semte.ch`. It requires responsible development in the core components, and we have needed to alter the frontend in order to support new ways of development. Most of this was due to JSONAPI not being standardised from the get go. Whether applications will turn out to be maintainable over longer periods of time is still to be proven.

6 Conclusion and next steps

We have shown that combining microservices with semantic technologies offers clean separation of concerns with decoupled microservices. We started using the platform in real-world scenarios to build web applications and share code across projects. It is our intention to further work out the kinks and make the platform more mature. We also want to provide more support to the user in the form of templates and addons. A user should get the platform up and running in just a few steps and be able to configure and extend the platform to his needs with minimal effort.

To make components reusable across projects, the microservices need to be cleary documented. Each microservice must specify its functionality, the REST calls it provides and the related frontend addons. Furthermore the used model and vocabularies should be documented as well. We are looking at frameworks like the OpenAPI Specification[23] and DOAP[24] to document the microservices in a structured way. The intention is to build a website that collects all microservices, the core ones as well as the ones built by the community.

A graph-store with a common ontology is well-suited for sharing information. However, the discovery of ontologies is not always that easy. The semantic web currently lacks a good search engine for discovering ontologies. Currently LOV and vocab.cc are available, but both services offer a rather limited search functionality. We intend to search further for solid ontologies for the core of this architecture. Ideally, an index of ontologies for commonly used services would exist in an easily searchable manner.

Data changes over time. It is important to recognize this reality. The platform does currently not keep track of history. Real-world use cases reveil however that versioning is oftentimes a requirement. The use of a versioned graph would allow us to merge graphs more easily and present a history of the system over time.

References

1. Ember.js: A framework for creating ambitious web applications. - `http://emberjs.com/` - 2016-03-12
2. AngularJS - Superheroic JavaScript MVW Framework - `https://angularjs.org/` - 2016-03-12
3. React Community - `https://github.com/reactjs` - 2016-03-12
4. mu.semte.ch - `https://mu.semte.ch/` - 2016-03-12

5. Microservices Architecture: The Good, The Bad, and What You Could Be Doing Better - http://nordicapis.com/microservices-architecture-the-good-the-bad-and-what-you-could-be-doing-better - 2016-05-01
6. Example voter application (backend) - Backend: https://github.com/madnificent/voter. Frontend: https://github.com/madnificent/voter-ui - 2016-05-01
7. JSON API - A specification for building APIs in JSON - http://jsonapi.org/ - 2016-03-12
8. Versteden A., Pauwels E., Papantoniou A., An Ecosystem of User-facing Microservices supported by Semantic Models - Position paper for the 5th International USEWOD Workshop, May 31st, 2015, Portoroz, Slovenia - http://usewod.org/files/workshops/2015/papers/USEWOD15_versteden_pauwels_papantaniou.pdf
9. Ember Data v1.13 released - https://emberjs.com/blog/2015/06/18/ember-data-1-13-released.html - 2016-05-01
10. mu-identifier: Core microservice for user identification - https://github.com/mu-semtech/mu-identifier - 2016-03-12
11. mu-dispatcher: Core microservice for dispatching requests to the preferred microservice - https://github.com/mu-semtech/mu-dispatcher - 2016-03-12
12. mu-semtech GitHub - https://github.com/mu-semtech - 2016-03-12
13. Docker - Build, Ship, and Run Any App, Anywhere - https://www.docker.com/ - 2016-03-12
14. Docker Hub - https://hub.docker.com/ - 2016-03-12
15. Docker Compose - https://docs.docker.com/compose/ - 2016-03-12
16. mu-project: Basis for constructing a new project on top of mu.semte.ch - https://github.com/mu-semtech/mu-project/ - 2016-03-12
17. Docker Swarm - https://docs.docker.com/swarm/ - 2016-03-12
18. Three-tier Architecture - https://www.techopedia.com/definition/24649/three-tier-architecture - 2016-05-01
19. What is the 3-Tier Architecture? - http://www.tonymarston.net/php-mysql/3-tier-architecture.html - 2016-05-01
20. Graphity - https://github.com/Graphity - 2016-05-01
21. Your Data Stories - H2020 project - http://yourdatastories.eu - 2016-05-01
22. Big Data Europe - H2020 project - big-data-europe.eu - 2016-05-01
23. The OpenAPI Specification Repository - https://github.com/OAI/OpenAPI-Specification - 2016-03-12
24. RDF schema for describing software projects - https://github.com/edumbill/doap - 2016-03-12