How will we interact with the #WebWeWant?
Position paper published at the 5th International USEWOD Workshop: Using the Web in the Age of Data, May 31st, 2015, Portoroz, Slovenia.

Invited paper

An Ecosystem of User-facing Microservices supported by Semantic Models

Aad Versteden, Erika Pauwels, Agis Papantoniou
TenForce, Leuven, Belgium

Abstract

Microservices promise to vastly simplify application complexity. Tiny applications offering very specific functionality are easy to build and maintain. But how do you keep your models in sync? Is there a way to share the microservices between parties? By leveraging the power of semantic technologies, we can create an ecosystem in which many microservices complete a unified model. A small semantic web to be used within an organization. This lets organizations have a taste of semantic technologies cheaply with minimal risk. Our strategy gives focus on supporting single page web applications, where javascript and REST API's are king. In this short position paper we describe an architecture for building, combining and sharing user-facing microservices for state-of-the-art web applications.

Keywords: microservices, user-facing, semantic model, REST API, RDF, SPARQL, triple store.

Introduction

Web usage [1] has evolved from pure data consumption to a "read-write" web, where the user can manipulate data. Where web pages used to be modified infrequently, content creation now goes hand-in-hand with content consumption. With the web being an easily accessible communication platform, more organizations are starting to use it as an application platform. The requirements of these applications tend to evolve over time. How can we keep up with the changing requirements whilst ensuring the code complexity stays contained? Can we ensure companies still have full control over what they share, if they want to embrace the semantic web? Is there a way in which we can share parts of the application with between related backends? Can we share domain-agnostic logic across unrelated applications?

Our proposed architecture, combining microservices[i] and the semantic web, provides answers to the above questions. Microservices are a key way of maintaining application complexity. Rather than one big monolithic application, many tiny applications are built. Each application has a limited specification. A service does one thing, and it does that one

thing well, much like a UNIX shell command. The general idea is that such scripts are easier to write, debug, and maintain.

In the light of the web becoming a data query and manipulation platform, we envision microservices querying and altering a specific type of information. Take user login and registration for example. One service could offer support for registering users, whereas another service could offer support for letting users login. The login service needs to know which users have registered. Multiple approaches for letting users register may exist. In order to freely combine these services, they need to manipulate a mutually shared model. Graph databases, like a triple store, contain flexible models which could be used by all services.

Ideally, microservices can be shared across various applications. Take the example of file upload. Many applications need some way to display images uploaded by the user. Few developers have interest in building yet another file upload implementation. Although libraries help a great deal, an ideal solution would share the complete microservice. Adding features to the application backend could become as simple as dropping in a new service. However, for a microservice to operate in a strange environment, we need to ensure it talks the right language. Whenever multiple actors consume and produce data in the same space, semantic technologies offer a great way out. By using well-known and standardized ontologies we ensure we are all talking about the same content in the same way.

Although a user-facing application may consist of many services, the user should not notice this. Simply sharing APIs is not sufficient for providing the end-user with a unified experience. We turn to the state-of-the-art of web application building. Single page web applications are becoming more and more common. Rather than letting a server construct the markup of a webpage, a single page application constructs the markup using the browser's Javascript runtime. These applications fetch their content from a web API, most often a JSON REST API. A single look andfeel can be provided by letting microservices output content in JSON format and using them in a single page application. Regardless of which microservice which offers a specific service.

A frontend should assume a specific API to program against. Which service offers that API should not be of a concern of the frontend. The microservices on the other hand may still need a way to identify the current user. By focussing on proven web technologies, we can build proxies for these constraints. One proxy will identify the user's session with a key unique for all microservices. Another proxy dispatches the user's request to the right microservice.

Our approach lets companies define and use ontologies for their business data without having the obligation to publish them online. Innovation towards the usage of the semantic web lies in lowering the barrier of entry. Organizations get to experience some of the benefits of the semantic web without publicly committing to it. We fully embrace non-semantic communication in the API, whilst making the backend fully aware of the semantic model. This vastly minimizes the risk of getting started with semantic technologies and greatly lowers the efforts for publishing linked open data later on. In short, a company can make use of these technologies, prepare, curate and link data for internal consumption and at a later stage decide which portion to publish as Open Data.

The paper presents and describes an architecture for building, combining and sharing user-facing microservices for state-of-the-art web applications. An architecture for which we are currently exploring practical deployments. Its structure is as follows: Section 2

provides an overview of the proposed architecture. In Section 3 the components of the architecture are further detailed and finally Section 4 concludes our paper with the next steps.

Architectural overview

The proposed architecture tackles the challenge of letting microservices cooperate easily for user-facing services. We see this architecture as a foundation component of a future framework which will support the development and integration of user-facing microservices, based on common web standards.

According to [2] it is preferred that the implementation of microservices receive as much freedom as possible. In practical cases for these services it is common to make technological choices to which each of the microservices should adhere. In our case we assume communication will be constructed from HTTP requests. We also make the technological choice that all meta-information should be stored in a shared triple store. In addition, the microservices should agree on shared vocabularies for using the data.

The proposed architecture [Figure 1] consists of four layers. The first layer presents the a visual interface to the user in the form of a single page application. The second layer is a support layer which selects the microservice that should handle the request and which identifies the user. The third layer consists of microservices which contain the bulk of the application. These layers maintain the business logic and manipulate the state of the application. The last layer consists of a storage layer in the form of a triple store.

The architecture, which is further detailed in [Figure 2] proposes three supporting systems. A system for user identification, one for dispatching, and a shared database. A fourth system which is not under the control of the system architecture, is the usage of a single page web application for the user interface. A user-facing request comes in from the single page application (1), is handled by the identifier (2) and forwarded to the dispatcher (3). The dispatcher picks the right service and lets it handle the request (4). The response follows the inverse path.

User identification is managed by the identifier. The identifier is a supporting service which captures all user requests. It identifies the user's session and ensures other services have structured access to it. It should be noted that the identification service does not add any information to the shared triple store itself. Identification is done by adding a UUID to the request and passing it along to the dispatcher. The specific implementation for identifying the user may change over time. As long as other services can depend on receiving the session identifier in the request header, they can continue to operate as expected.

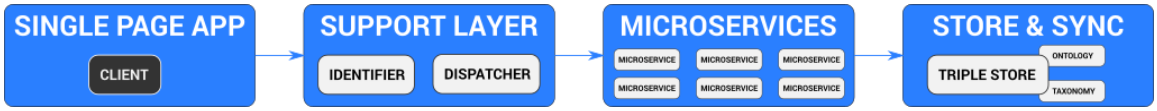

Figure 1: Architectural Overview

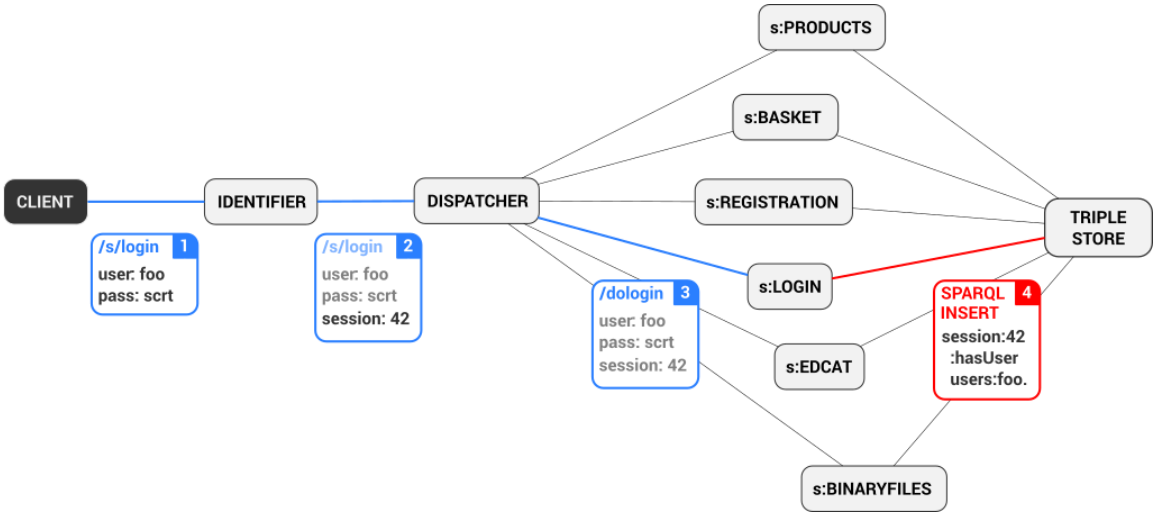

Figure 2: Components of the architecture

Selecting the right microservice is handled by the dispatcher. It is possible that multiple microservices offer the same service. Multiple frontends may be present on one ecosystem of microservices. The dispatcher ensures we can select the right microservice for a specific request. For this, it analyzes the request path and translates it to a target path on a specific microservice. Although the dispatcher may alter the request path, the intention is that it never translates the request itself to a different request. The interface offered by the microservice needs to follow the standards the frontend expects.

In order to manage forwarded HTTP requests, the microservice itself is a tiny web service with its own web server. It receives the request, performs the necessary queries on the joined triple store and returns the response for the user. The response propagates back through the dispatcher and identifier to the single page web application. For access control, the microservice uses the session identifier to find information of the user in the triple store. A login service would connect this session id to a specific user in the database. An example basket service could verify that the user is logged in when she needs to confirm her basket.

The single page web application needs to communicate with the service by using the JSON REST interface offered by the microservices. Although there is no requirement on the specific interface, it is advised for the services to publish using common JSON formats like JSON API [4]. Hosting the single page web application can be handled by a simple static host. It would seem that this construction limits the consumers to smart javascript-enabled clients but great efforts are being made to host these applications as static pages. In this respect we see the evolution of ember-cli-fastboot [3] as an important step. With this approach we can offer a fluent unified user-experience whilst still supporting older clients.

The broad architecture uses a support layer of microservices to augment the request's information and to delegate the request to the designated service in the microservice layer. This construction gives the services a great deal of freedom, but supports them in their user-facing tasks.

The components in detail

The proposed architecture can offer a lot of flexibility on many aspects, due to rigid constraints on other aspects. In this section, we discuss the microservices in the support layer first and continue on with some typical user-facing microservices. We conclude with a summary on how this architecture presents itself to a single page application.

Session identification

The user's session needs to be uniquely identified throughout the ecosystem, regardless of the microservice which handles the request. Ideally, a microservice should be able to identify the session with knowledge of only the HTTP protocol and the triple store. We present a separate proxy to handle session identification which places the session in an HTTP header.

Each request which needs to be passed to a microservice first hits the identifier microservice in the support layer. This microservice identifies the current user's session. The session is identified in the custom `MU-SESSION-ID` HTTP message header. As each service is effectively a mini web service on its own, hence it needs to understand the HTTP protocol. The identification thus places no additional constraints on the microservices.

Any microservice can use the session identifier as the basis for connecting information to the session in the triple store. Traditional session variables can therefore be added as triples using the session identifier as subject. With this construction, the session is placed in a central location understood by all services. Furthermore, the backing ontologies ensure all services can augment the session store with information they understand about the user without causing conflicts.

We have made a draft implementation of the user identification service. There is no reason why the user identification service could not be swapped out with a different service. The specific implementation may vary over time. To our understanding, the user identification service may have many long-running connections. It should therefore be implemented in a lightweight manner. Our draft implementation is written in Elixir, a language which runs on the same virtual machine as Erlang and with similar performance characteristics. A single Erlang microservice can handle many long-running connections without placing much stress on the hardware resources. This makes it well-suited for this purpose. The current implementation uses a session cookie to store the user identifier in a way in which the user can't tamper with the session itself. In case of scaling issues, we could scale this service horizontally.

The user identification is a cornerstone in the microservice ecosystem. It ensures all user-facing microservices have a common understanding of who the user is. It does so in a way which does not add extra constraints on how the microservices need to be implemented.

Request dispatching

The second service a request hits is the dispatching service. This service is much like the router in many web MVC frameworks (eg: ASP.NET MVC [5], Ruby on Rails [6]). For each request made by the user, the dispatcher decides which microservice to contact, and on which URL. It is our intention that this service alters at most the URL of the request, namely the base and the path of the URL.

The network of microservices could be used to host more than a single user-facing service. We foresee a backend architecture with many microservices, offering multiple API's for various frontends. Perhaps some microservices should only be accessible on an internal API. Which services are available in an API should therefore be a conscious decision. The request dispatcher ensures only the necessary services are published on a specific endpoint.

Different microservices may fulfill the same need, perhaps with different performance characteristics. Due to the different performance characteristics, it may be that multiple microservices are running which fulfill the same need. The dispatcher should ensure that these microservice are published on the right endpoint. For this reason, the dispatcher must decide the path on which a service is published. The dispatcher performs minimal changes to the request URL so it is understood by the microservice.

As the dispatcher receives many long-running connections, we have opted to make the prototype in the same platform as the session identification service. The Elixir language offers an expressive language whilst offering good performance characteristics. The prototype implementation dispatches requests to the designated microservice. This operates as a traditional web server which uses the Plug Router to dispatch the routes. The Plug Router offers sufficient support dispatching requests, but administrators may prefer a higher level language or a configuration file to specify what is published, and where. Alternatives can be built within the same set of constraints.

The dispatcher is the second core element of the architecture. It provides a minor abstraction over the microservices, deciding where they are accessible.

Login & registration

The Login service and the Registration service allows users to identify themselves. Login information is stored in the triple store so other services can act on the logged in user. The Login service and the Registration service are two separate services, as they may come with different features and constraints.

Some services may require to connect specific actions to specific users. Perhaps only an administrator may view sensitive information. And it may only make sense to order a product if the user who has made the order is verified. Identifying a user is handled by the Login service. Creating new users may be handled by the Registration service. Both services are drafts, more advanced implementations may be preferred.

A simple example should express the prototypical case of a logged in user is [Figure 3]. The grey zone of the picture is generated by the registration service. The blue content is generated when the login service identifies the user. Let's look at each of them in more depth. 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, they can evolve separately.

The Registration Service receives input from the user. It constructs the user, and attaches the specific account to the user. A user may have multiple accounts by which he can connect to the application. In our current setup, user login is rudimentary. A User is generated separately from the User Account. Both are connected through the `foaf:account` predicate. A registration service may evolve by adding a requirement for email confirmation. Furthermore, the registration service may also understand registration by use of Facebook connect or similar.

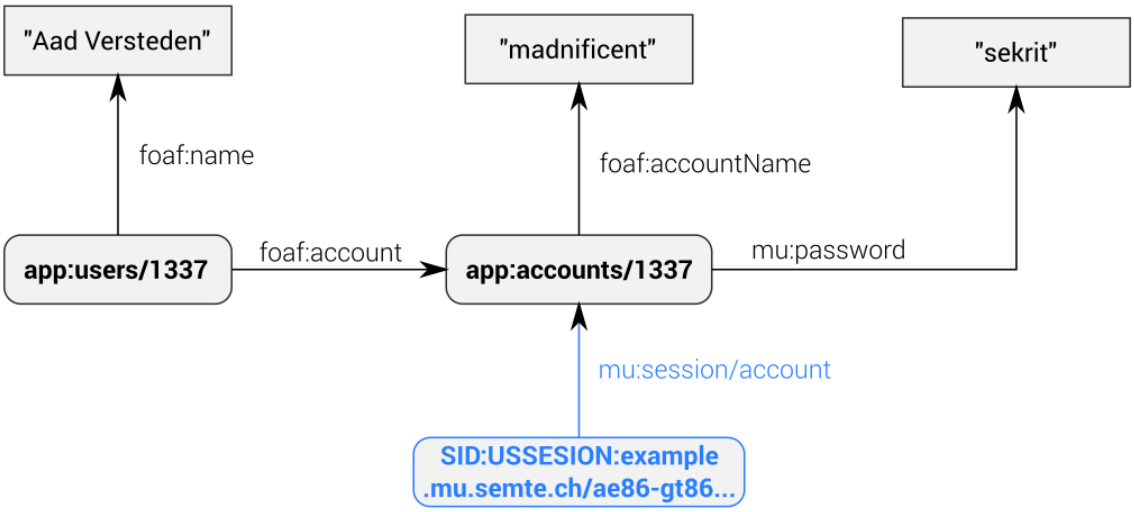

Figure 3: User login example

The Login Service handles the actual login of the user. When a user tries to login, the service receives the user's username and password. It searches the database for an account which matches both the username and the password. If the combination of both exists, it connects the session URI (which is found in the `MU-SESSION-ID`[ii] request header) to the user account using the `` `mu:session/account` `` predicate. We defined this predicate by the lack of finding an appropriate commonly accepted standard, it is subject to change. Other services can now find the user by searching for the following pattern in the database:

```
?user foaf:account ?account.
<SESSION-URI> mu:session/account ?account.
```

Although the Login Service and the Registration Service are developed separately, they obviously work in tandem to provide a complete service. Due to the different loads that both services may need to handle, they could be written in a different language with different constraints. Our architecture allows both services to be offered as two separate services, or in a single service. Our draft implementations of both services are written in Ruby with the Sinatra web framework. Ruby is an expressive language and Sinatra is a light-weight web framework for it. This framework makes for code that is easy to read, even though it may not be the fastest to execute. As login and registration are actions which a user performs only once during a session, we found the benefits of the simplicity to outweigh the downsides of the scaling.

The Registration Service and the Login Service may change during the life of the application. The used FOAF model and the user data would not need to be changed. Due to the use of common standards, other microservices depending on the logged in user won't need to be altered.

File upload

The file upload is a prototypical service which many applications 'just need'. It offers a nice example of where to use the triple store, and where to leave it be. A file upload service

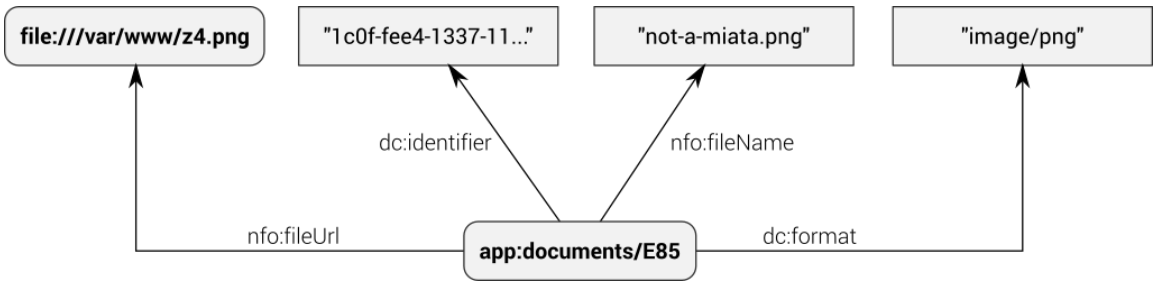

Figure 4: File upload example

needs to manage binary files. Users, or administrators, can upload content to the site and expect it to be available. What constitutes a file service, and what can we do with it?

Our File Upload service offers create, read and delete operations (update is not support in the current version). Although all content could be stored in the triple store, we opt to use technologies pragmatically. In terms of files, that means storing the metadata of the file in the triple store, and storing the binary content in a traditional filesystem.

Once a file is uploaded, it is stored on the filesystem. Basic metadata about the file is stored in the Nepomuk File Ontology. The actual file is stored in an appropriate location on the file system. Basic file system abstractions hold regarding the storage of the raw file.

The File Upload service offers only basic functionality. However, with the metadata being stored in the filesystem, other services can access and operate on the same content. An image resizing microservice can use the same metadata to offer new services to the client. This service is completely decoupled from the File Upload service aside from the used ontologies. It acts and behaves as one.

The File Upload service indicates that the backing triple store can be used in a hybrid fashion. With resources referring to external locations, services may interact with various types of content. It also shows highly reusable microservices can be offered and combined as necessary.

Conclusion and next steps

We have shown that combining microservices with Semantic Technologies offers clean separation of concerns with nicely decoupled microservices. Furthermore, it allows organizations to taste from two ground-breaking technologies within a limited scope, thereby changing the risks of trying out semantic technologies. Although this architecture is backed by real-world needs and a real-world implementation, it is our intention to further work out the kinks and explore this architecture in more real-world situations.

The architecture offers other benefits which we intend to explore, as well as some caveats which we intend to tackle with more experience. The low hanging fruit hangs in interchanging information. Caveats are found in the configuration of the services and the user of more common ontologies. The lack of a great search engine for ontologies is greatly hindering in this respect.

Analyzing the day-to-day usage logs of this new architecture will allow us to gain new insights about the advantages and challenges of semantic technologies in practice. Log analysis can profit from the semantics of the underlying data and processes, and the observed and mined patterns can be leveraged for improving the composition of the microservices.

The deployment of microservices is a story on its own. We are currently using docker[9] for running each of them. The composition and deployment of docker containers is under active development in the docker community with the docker-compose[10] framework. We are following up on these developments to ensure the deployment strategy is solid. We intend to further document a sensible deployment strategy as the docker community matures.

A graph store with a common ontology is well-suited for sharing information. With this architecture we make it easy for organizations to share parts of their knowledge graph between various systems. Be it systems within the organization or across many organizations. This lets companies have a taste for the power of the semantic web without having to dive into it completely. We intend to provide a service which exports some of the content in the triple store as Linked Data Fragments [7], further aiding linked data adoption.

We further intend to explore the route in which data changes over time. The semantic web has an open world assumption. Everything is assumed to be true. However, some data may change over time. Within the context of a company, it is important to recognize this reality. The use of a versioned graph would allow us to merge graphs more easily and present a history of the system over time. The idea of versioned data is one of the cornerstones of the lambda-architecture and we intend to discover if the benefits hold for our implementation.

Last, but certainly not the least important is the discovery of solid ontologies for expressing the reality in the triple store. Ironical for the semantic web, we lack a good search engine for discovering solid ontologies. 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.

References

[1] Tatnall A. eds., Web Technologies: Concepts, Methodologies, Tools, and Applications, IGI Global, October 2009

[2] Newman S., Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, February 2015.

[3] tildeio/ember-cli-fastboot - https://github.com/tildeio/ember-cli-fastboot - 2015-03-30

[4] JSON API :: A standard for building APIs in JSON. - http://jsonapi.org/ - 2015-03-30

[5] ASP.NET MVC Routing Overview (C#) - http://www.asp.net/mvc/overview/older-versions-1/controllers-and-routing/asp-net-mvc-routing-overview-cs - 2015-03-30

[6] Rails Routing from the Outside In - http://guides.rubyonrails.org/v4.1.8/routing.html - - 2015-03-30

[7] Linked Data Fragments - http://linkeddatafragments.org - 2015-03-31

[8] Session Identification URI - http://www.w3.org/TR/WD-session-id - 2015-03-30

[9] Docker - Build, Ship, and Run Any App, Anywhere - https://www.docker.com/ - 2015-04-11

[10] Docker Compose - Docker Documentation - https://docs.docker.com/compose/ - 2015-04-11

Used ontologies

[DC] Dublin Core - http://purl.org/dc/terms
[FOAF] Friend-of-a-friend - http://xmlns.com/foaf/0.1/
[MU] Semantic Microservices Ontology - http://mu.semte.ch/vocabulary/
[NFO] Nepomuk File Ontology –
http://www.semanticdesktop.org/ontologies/2007/03/22/nfo

---

[i] http://en.wikipedia.org/wiki/Microservices

[ii] We use MU-SESSION-ID rather than SESSION-ID as the latter is used by the Session Identification URI working draft [8]. We represent the MU-SESSION-ID resource in a similar format as the SESSION-ID.