A Reference Architecture for Real-Time Microservice API Consumption

Cristian Gadea Mircea Trifan
Dan Ionescu
School of Electrical Eng. and Computer Science
University of Ottawa
Ottawa, Ontario, Canada
cgadea, mircea, dan@ncct.uottawa.ca

Bogdan Ionescu
Mgestyk Technologies Inc.
Ottawa, Ontario, Canada
bogdan@mgestyk.com

Abstract

Modern web frameworks and backend-as-a-service providers make it possible for real-time updates to a NoSQL data model to be reflected in the user interfaces of multiple subscribing end-user applications. However, it remains difficult for users to dynamically discover and instantly make use of the data provided by the plethora of REST APIs in existence across various cloud providers today. This paper presents a reference architecture built on the idea of a scalable NoSQL database that allows multiple subscribers to receive instant notifications of database changes through the use of a "livequery". By keeping one WebSocket connection open between each client web browser and an Object Synchronization Server, this paper shows how data from multiple disparate REST APIs can be organized and transmitted to interested clients via the database. An example is given featuring a collaborative rich-text editor that makes use of a Named-Entity Recognition microservice.

Categories and Subject Descriptors D.2 [*Software*]: Software Engineering; C.2.4 [*Distributed Systems*]: Distributed Applications

Keywords microservices, docker, cloud computing, collaborative editing, real-time web, named entity recognition

1. Introduction

Microservices, as described in (Matthias and Kane 2015) and (Newman 2015), are a new type of software architecture that composes an application as a collection of independent individual services. Each service is dedicated to a single business capability. Services can be implemented in many programming languages and can be deployed automatically. Microservices allow for flexible deployments, as well as shorter development times, giving smaller teams complete ownership of vertical slices of related developments randing from the databases to the user interface.

Docker can be used to implement the microservices architecture by introducing software containers that enclose deployed applications. This is referred to as "container based virtualization" or "operating system level virtualization" (as opposed to "hardware virtualization" provided by traditional virtual machines). The advantages of using Docker include rapid application deployment, portability across machines, component reuse, version control, minimal overhead, a lightweight footprint, sharing, and simplified maintenance.

Alpers et. al. (Alpers et al. 2015) describe a microservice-based implementation of a business process modeling tool comprising a collaborative Petri Net editor accessible using two interfaces: a web client and a native mobile application. A comparison between a three-tier architecture and the new microservice approach is also provided. (Abdelbaky et al. 2015) proposes a prototype framework that allows Docker containers to be deployed across multiple clouds and data centers. It uses a constraint-programming model for selection and utilization of the resources. In order to compose multiple microservices with different APIs and perform service discovery, a fully decentralized open source solution based on the Serf project is described in (Stubbs et al. 2015). (Toffetti et al. 2015) is a position paper that proposes a novel architecture for scalable and resilient self-management of microservices applications on cloud. However, these implementations do not address the issue of synchronizing data from microservices on different clouds to multiple subscribing web clients.

The "Notebook Microservice And Swagger" implementation from IBM (not) shows how the creation of microservices is increasingly approachable. The project uses Swagger, a REST API code generator (swa), to describe a REST-based microservice. Code generation is then used to synthesize a Jupyter Notebook (jup) that exposes the REST services from within a Docker container. The architecture described in this paper will allow for the real-time discovery and consumption of such microservices, as will be shown using a collaborative rich-text editor.

The inclusion of microservices to assist the document editing process was recently explored by the New York Times (nyt). The "NY Times Editor" allows journalists to tap into the power of machine learning algorithms by applying tagging and annotations as the users type new words. The additional information that is brought in from the microservice augments the text and assists the authors in real-time. The microservices-based editor was found to be a welcome change from their previous monolithic content management system (CMS). However, their prototype used a simple web-based text editor that did not have any collaborative features, and changing microservice subscriptions at run-time was not explored.

This paper presents a reference architecture that makes data from multiple microservices available to web-based clients in real-time through the use of a NoSQL database. REST APIs can be man-

CrossCloud'16 April 18, 2016, London, United Kingdom

ACM 978-1-4503-4294-0/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2904111.2904115

aged and exposed to web clients, who can then access the microservices and receive their responses through a "livequery" mechanism whereby updates to database data are instantly communicated to subscribing clients. A collaborative rich-text editor will be used to demonstrate the architecture by accessing a Named-Entity Recognition microservice that enhances the editor contents.

The remaining sections of this paper are as follows: Section 2 presents the reference architecture. Section 3 then provides the implementation details for this architecture. Section 4 discusses the resulting system. Finally, Section 5 provides concluding remarks and avenues for future work.

2. Architecture

The architecture proposed in this paper aims to allow web clients to subscribe to database changes that indicate the availability of REST APIs. Database changes are also used to perform data requests and to receive the REST responses. A *Microservices Management Server* is used to coordinate the execution of the REST requests on behalf of the client. Figure 1 shows a diagram of the proposed architecture. The purpose of each component is discussed in this section.

As the real-time nature of the architecture is heavily dependent on a NoSQL database, it is important to clarify a few key points. The NoSQL terminology used in this paper assumes that a "collection" consists of multiple "documents". A collection is therefore similar to a SQL table, while a document is similar to a row within a table. The proposed architecture makes use of the following three collections:

1. `apiRegistry`: This collection is used by the *Microservice Management Server* to store the descriptions of available REST microservices. It is also used by a *Web Client* to receive notifications as descriptions of microservices are added or removed.
2. `serviceCalls`: A document is added to this collection by a Web Client that wishes to create a call to a REST microservice. The document contains details about the call to be made, as well as the name of the collection to be used as the destination for the responses from the REST call (for example, a collection named `results`).
3. `results`: This collection is accessed by a *REST API Adapter* or a *Adapted Microservice with REST API* and is used to store the responses of a REST call, as requested and specified by a *Web Client*.

2.1 Object Synchronization Server & NoSQL Database

At the heart of the proposed architecture is an object synchronization mechanism that is capable of keeping multiple web clients up-to-date based on real-time synchronization of the local browser data model with that of a real database. *Web Clients* store data locally in the browser as JavaScript objects and, whenever possible, they synchronize their local *Mini Database* against a full-fledged *NoSQL Database*. The offline version of the data model is particularly important for supporting intermittent connections of mobile devices, allowing the data to synchronize whenever connectivity is available. The NoSQL nature of the database is essential for providing the scaling, sharding and replication functionality expected of modern architectures, as well as to better support hierarchical data required for collaborative document editing, for example.

A publisher-subscriber mechanism is applied so that multiple clients can be subscribed to the data and receive real-time updates as the data changes in the database. Synchronization therefore includes both updates sent to the database when the data model changes on the client side (such as due to user interaction), as well as updates sent from the database as "push" notifications to all subscribed clients whenever the model changes in the database (such as the result of another user's changes).

PouchDB is an example of such an object synchronization system that uses CouchDB as its back-end database (pou). The Meteor web framework (met) makes use of a "livequery" mechanism on top of MongoDB to notify subscribers of model updates. Backend-as-a-Service providers such as Firebase (fir) and databases such as RethinkDB (ret) are also designed to provide such bi-directional synchronization capabilities to JavaScript clients.

In addition to basic object synchronization, algorithms such as differential synchronization (Fraser 2009), Commutative Replicated Data Types (CRDTs) (Shapiro et al. 2011) or Operational Transformations (Davis et al. 2002) can be implemented by the Object Synchronization Server to allow individual text characters or hierarchical elements to be exchanged and merged while still ensuring all users reach a consistent state of the data. Such functionality is required to enable conflict-free collaborative editing scenarios similar to Google Docs (gdo 2009).

2.2 Microservice Management Server & Manager UI

With the use of the *Microservice Management Server* and its real-time livequery subscription to the NoSQL server, the *Microservice Manager User Interface (UI)* provides users with tools for managing REST API microservices by representing active microservices using an API description language such as the RESTful API Modeling Language (RAML) (RAM) or Swagger (swa). A text editor allows defining existing microservices using the description language. Each description is submitted (or synchronized in real-time) as a document to a `apiRegistry` collection in a *NoSQL Database*. The *Microservice Manager UI* can be implemented as a web client or regular desktop application.

2.3 Web Client

The *Web Client* contains the domain-specific user interface and functionality to meet the requirements of the system by making use of available microservices. It contains a *Mini Database* model layer in JavaScript that is synchronized with the *Object Synchronization Server*. It connects to the *Object Synchronization Server* using a unique identifier (for example, a username) and, as much as possible, maintains an active WebSocket connection to the *Object Synchronization Server*. It subscribes to the `apiRegistry` collection of APIs and presents the available functionality to the user through user interface elements such as lists and buttons. Note that buttons can be added and removed dynamically based on the latest state of the `apiRegistry` collection, as configured via the *Microservice Manager UI*.

Based on the API information received by the *Web Client* from the `apiRegistry` collection, the *Web Client* may request a call to a REST microservice. Such calls are requested by creating an entry (document) in a NoSQL database collection named `serviceCalls`. The *Microservice Management Server* is always subscribed to this collection as it awaits requests from the *Web Client*. All service call documents made by the *Web Client* also specify a target `results` collection (the collection name may include the user's identifier and API identifiers). The *Web Client* then subscribes to this collection and awaits a push-based response to the placed service call, to be created by the *REST API Adapter* of the requested REST API, or by a *Adapted Microservice with REST API*. Figure 2 provides a high-level summary of this sequence of events.

The *Web Client* should present session information to the user such as a list of which other users are online. Text chat is another basic collaboration feature that should be provided. The chat can have its state synchronized using the offline storage mechanism that is synchronized to the *Object Synchronization Server*. Interactions

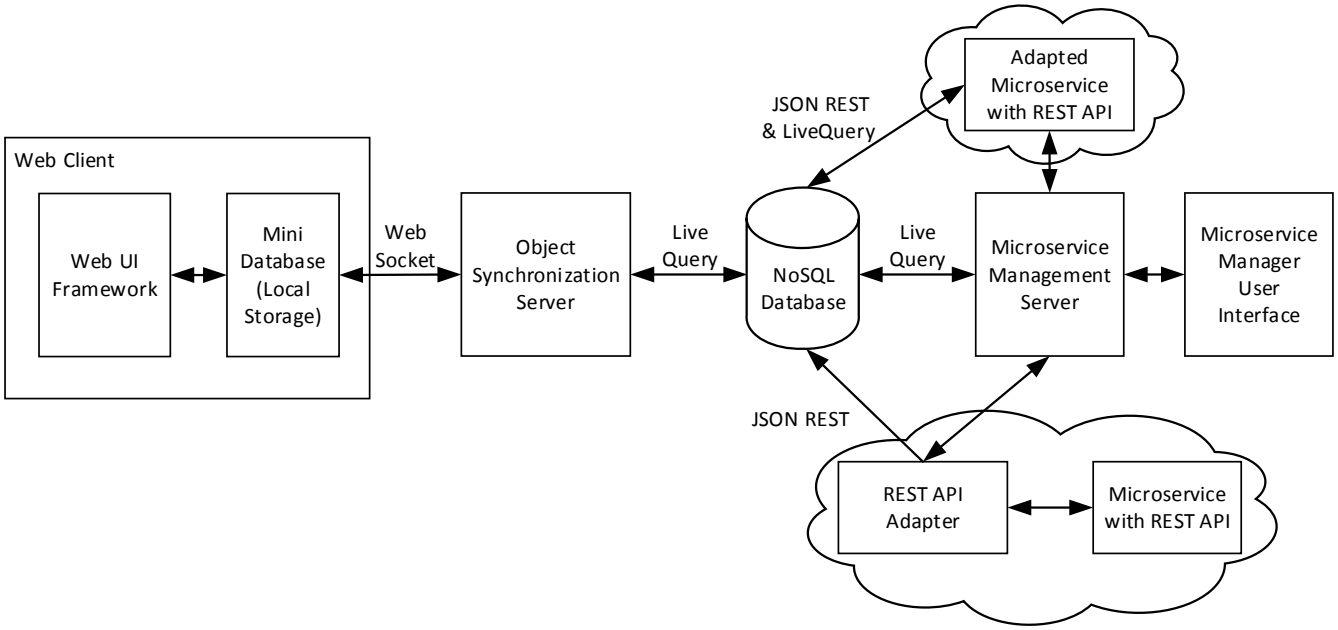

Figure 1. Cross-cloud API architecture.

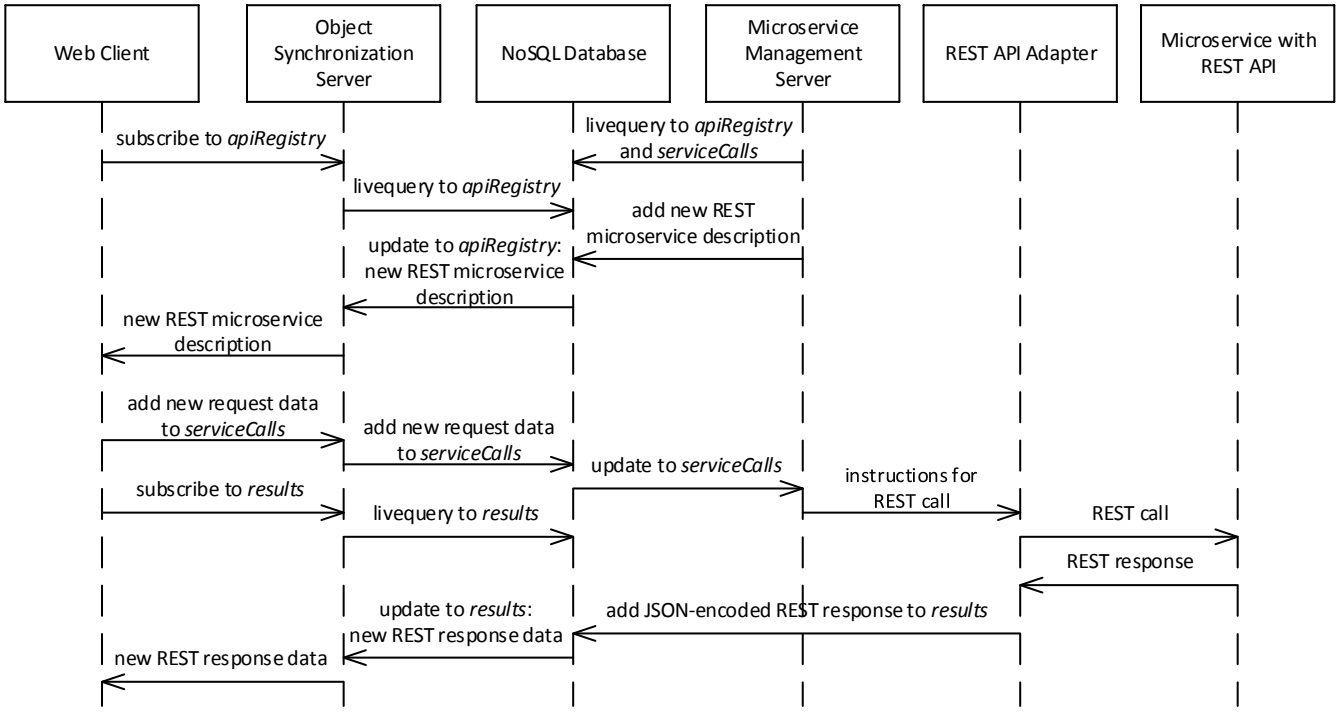

Figure 2. Sequence of events for a Web Client to receive push-based data from a new REST microservice.

with the *NoSQL Database* must be done using the livequery approach that ensures all clients are subscribed to receive notifications to any of the data structures relevant to them. The *Web Client* can make use of *Web UI Frameworks* such as Angular (ang) or Bootstrap (boo) to create a pleasant user interface for these elements.

2.4 Adapted Microservices & REST API Adapter

The *Microservice Management Server* retains a real-time subscription to the `serviceCalls` collection. As new requests appear from different *Web Clients*, the *Microservice Management Server* performs the indicated GET or POST calls to the requested REST-based microservice. Rather than calling the microservice directly, however, the *Microservice Management Server* instructs a *REST API Adapter* component to perform the call and to publish its response directly into the `results` collection specified by the client. Existing REST services or microservices can therefore be accessed by such a *REST API Adapter*, including existing Docker or AWS Lambda-based (lam) containers from across different cloud providers (organized and managed via the *Microservice Manager UI*). The *REST API Adapter* can also itself be deployed as a microservice.

To best support the proposed architecture, new microservices should be developed to accept the parameter containing the name of the specific `results` collection and should publish the REST response data to that collection directly. Such a microservice is denoted as *Adapted Microservice with REST API* in Figure 1. In addition, depending on its purpose, the microservices may create a livequery-based subscription to a different collection in the NoSQL database containing data that may be required for the microservice to perform its duty (for example, monitor a text document that is being edited in real-time).

Depending on the type of service, additional intelligence can be applied by the Microservice Management Server to subscribe clients to existing `results` collections if the data generated by the microservice is not specific to one user (for example, multiple users accessing a microservice providing a real-time stream of latitude/longitude values of a moving vehicle).

3. Implementation

The closed-source implementation of the proposed architecture involved a rich-text editor that is enhanced using additional microservices. The document content is synchronized among multiple session participants, but each participant may wish to access different microservices as they are editing.

The REST API microservice that was implemented performs Named-Entity Recognition (NER) on any given text. NER identifies POLE data: People, Organizations, Locations and Events. The Stanford NER components (Finkel et al. 2005) are used as the foundation of the implemented microservice.

The components of the architecture were implemented as microservices using a variation of the `docker-compose` file shown in Listing 1.

Listing 1. Contents of the docker-compose file for the implemented architecture.

```
mongodata:
  image: mongo:latest
  volumes:
    - /data/db
  command: --break-mongo

mongo:
  image: mongo:latest
  volumes_from:
    - mongodata
  ports:
    - "27017:27017"
  command: --smallfiles

nerserver:
  build: NERServer
  ports:
    - "8080:8080"

objectsyncserver:
  build: ObjectSynchronizationServer
  ports:
    - "8070:8070"

managementserver:
  build: ManagementServer
  ports:
    - "8060:8060"

webserver:
  build: CollaborationClient
  ports:
    - "8090:8090"
```

A custom *Object Synchronization Server* was developed in Node.js to provide the previously-described livequery functionality on top of a MongoDB *NoSQL Database*. The server supports XML-based Operational Transformations to allow for collaborative rich-text editing by synchronizing specific elements of the Document Object Model within the web browser. The *Microservice Management Server* was also written as a Node.js server with similar subscription capabilities.

Swagger (swa) was selected as the REST API description language. Swagger defines an open standard for representing REST interfaces using a YAML ("YAML Ain't Markup Language") syntax. The indentation based scoping of YAML makes it more user-friendly to compose than JSON. For example, it is simple to generate the REST APIs for a microservice that returns Stanford NER annotations. The text to be annotated is given as input and the replies can be either JSON or XML, as specified using the YAML description language. Swagger converts the API description from YAML to JSON for persisting in the `apiRegistry` collection in MongoDB.

The Swagger editor UI, acting as the *Microservices Manager UI* for this implementation, is shown in Figure 3. A YAML code editor is presented on the left side of the interface, while testing functionality and other information is available via the right side.

An example of the YAML file for the generated NER microservice is shown in Listing 2. A text is given as an input parameter for the "entities" endpoint and the microservice returns the annotations.

Listing 2. YAML for NER REST Microservice.

```
swagger: '2.0'
info:
  version: "0.0.0"
  title: Stanford NER
paths:
  /entities/{text}:
    parameters:
      - name: text
        in: path
        description: The text to be annotated.
        type: string
        required: true

    get:
```

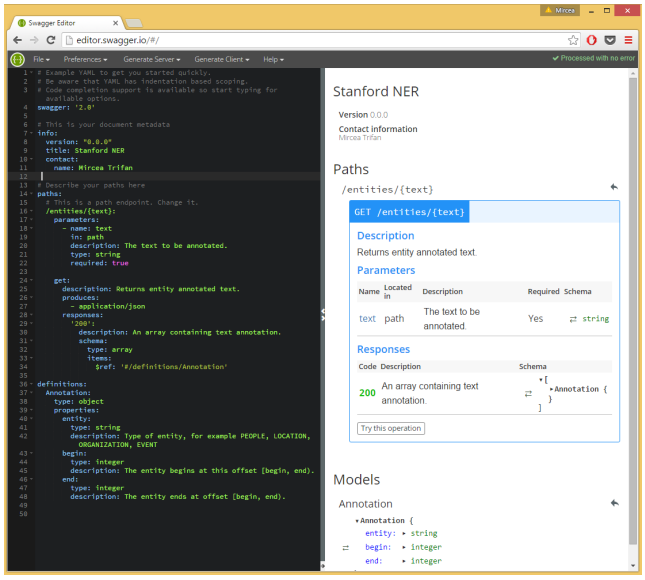

Figure 3. The Swagger editor is inspiration for the Microservices Manager UI.

```
    description: Returns entity annotated text.
    produces:
      - application/json
    responses:
      '200':
        description: An array containing
          text annotation.
        schema:
          type: array
          items:
            $ref: '#/definitions/Annotation'

definitions:
  Annotation:
    type: object
    properties:
      entity:
        type: string
        description: Type of entity, for example
          PEOPLE, LOCATION, ORGANIZATION, EVENT
      begin:
        type: integer
        description: The entity begins at this
        offset [begin, end).
      end:
        type: integer
        description: The entity ends at
        offset [begin, end).
```

4. Results

The implementation presented in Section 3 results with a collaborative rich-text editor capable of highlighting words as the user is typing if the NER Microservice recognizes them as entities. This is made possible by having the user enable the NER feature through a button in the user interface, which causes the NER Microservice to begin publishing to the MongoDB collection requested by the *Web Client*. The microservice is also able to receive updated rich-text editor content in real-time by creating a MongoDB livequery to the collection in which the editor's content is stored.

An example output of the REST-based NER Microservice for a document containing the words "John Smith was born in California" is given in Listing 3, correctly identifying the "people" entity of "John Smith" between character positions 0 and 10, and the "location" entity of "California" between character positions 23 and 33. The JSON-encoded response is then added to the `results` collection specified by the client. Once added, the client instantly receives the updated response values and displays them accordingly.

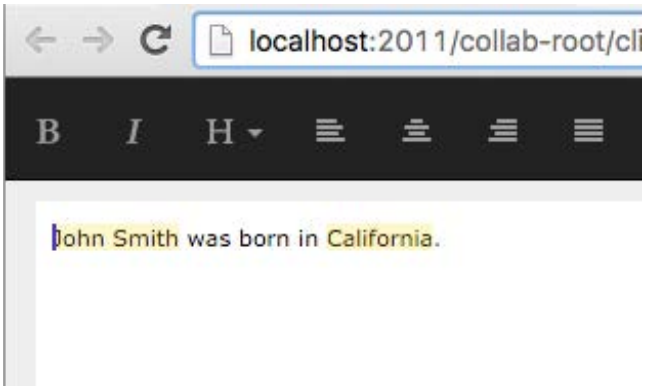

Figure 4. Collaborative rich-text editor supported by NER microservice.

Listing 3. Sample REST Output of NER Microservice.

```
{
  "code": 4,
  "type": "ok",
  "entities":[
    {"entity":"PEOPLE", "begin":0, "end":10},
    {"entity":"LOCATION", "begin":23, "end":33}
  ]
}
```

Figure 4 shows what the users see as they are typing. The two words have been highlighted as a result of the values returned by the NER Microservice in real-time. This is a good example of an API that is unrealistic to run on the web-based client side due to its computational implications, yet by accessing it as a real-time microservice through the proposed framework, the user is able to instantly benefit from the server-side processing.

5. Conclusion

This paper presented a reference architecture that makes use of a "livequery" technique whereby web-based clients are instantly notified of updates to data within a NoSQL database to which they subscribed. A Microservice Manager UI was used to organize multiple REST APIs and allow web clients to receive the responses from REST microservices through one WebSocket connection to an Object Synchronization Server. It was demonstrated how a collaborative rich-text editor can benefit from accessing an external Named-Entity Recognition microservice, which was introduced to the system through Swagger's YAML-based description language. The microservice returned JSON data to the client via a database collection to which the client was subscribed. Future work will consist of introducing additional APIs hosted on different cloud providers to the system and performing a more thorough evaluation of the scalability and performance of the various components of the architecture.

References

RAML. URL `http://raml.org/`. [Accessed: March 2016].

AngularJS - Superheroic JavaScript MVW Framework. URL https://angularjs.org/. [Accessed: March 2016].

Bootstrap The world's most popular mobile-first and responsive front-end framework. URL http://getbootstrap.com/. [Accessed: March 2016].

Firebase: Build Extraordinary Apps. URL https://www.firebase.com/. [Accessed: March 2016].

Apache Jupyter. URL http://jupyter.org/. [Accessed: March 2016].

AWS Lambda — Product Details. URL https://aws.amazon.com/lambda/details/. [Accessed: March 2016].

Meteor: A better way to build apps. URL https://www.meteor.com/. [Accessed: March 2016].

Notebook Microservice And Swagger. URL http://blog.ibmjstart.net/2016/01/28/ notebook-microservice-and-swagger/. [Accessed: March 2016].

NYTimes R&D Labs Editor. URL http://nytlabs.com/projects/editor.html. [Accessed: March 2016].

PouchDB, the JavaScript dabatase that Syncs! URL http://pouchdb.com/. [Accessed: March 2016].

RethinkDB - The Open-Source Database for the Realtime Web. URL https://www.rethinkdb.com/. [Accessed: March 2016].

Swagger. URL http://swagger.io/. [Accessed: March 2016].

Google Docs, 2009. URL http://docs.google.com. [Accessed: March 2016].

M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder. Docker containers across multiple clouds and data centers. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 368–371, Dec 2015. doi: 10.1109/UCC.2015.58.

S. Alpers, C. Becker, A. Oberweis, and T. Schuster. Microservice Based Tool Support for Business Process Modelling. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International*, pages 71–78, Sept 2015. doi: 10.1109/EDOCW.2015.32.

A. H. Davis, C. Sun, and J. Lu. Generalizing Operational Transformation to the Standard General Markup Language. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, pages 58–67. ACM, 2002.

J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 363–370. Association for Computational Linguistics, 2005.

N. Fraser. Differential Synchronization. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 13–20. ACM, 2009.

K. Matthias and S. P. Kane. *Docker: Up & Running*. O'Reilly Media, Inc., 2015. ISBN 1491917571.

S. Newman. *Building Microservices*. O'Reilly Media, Inc., 2015. ISBN 1491950358.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

J. Stubbs, W. Moreira, and R. Dooley. Distributed systems of microservices using docker and serfnode. In *Science Gateways (IWSG), 2015 7th International Workshop on*, pages 34–39, June 2015. doi: 10.1109/IWSG.2015.16.

G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pages 19–24. ACM, 2015.