Intercepting HTTP Requests for a Fast and Secure Two Tier Architecture

Mathieu Nayrolles, Abdelwahab Hamou-Lhadj
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
{mathieu.nayrolles, wahab.hamou-lhadj}@concordia.ca

Marc Montagne, Vincent Satiat
Toolwatch
Lausanne, Switzerland
{marc, vincent}@toolwatch.io

***Abstract*—zsd**

***Index Terms*—Web Paradigm; Web Architecture; Two tier architecture**

I. Introduction

Modern web development is dominated by the three-tier architecture. In the three-tier architecture, the client (i.e., the browser) presents the user interface which has been generated according to business rules (i.e., dynamic web-page generation). The business rules might rely on stored date (i.e., a database) [1]. Each tier (presentation, application, and data) is developed and maintained separately as interchangeable modules with well-defined interfaces. This provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application [2]. In addition to the flexibility during development and maintenance, the three-tier architecture also provides flexibility during operation. Indeed, each tier can be scaled separately to meet the demand. Auto-scaling such architecture (i.e, adjusting the computational power of each tier without service interruption) is a popular area of research [3]–[6] and business [7], [8].

While the three-tier architecture is popular and provide considerable advantages for developers and maintainers both, the model had been introduced more than twenty years ago. Despite the fact that this architecture has been refined and built upon during the last two decades, the fact remains that, it has been theorized at a time where the Internet user base was 44 million [9], Netscape was dominating the browser market [10], Javascript [11], Java [12], Linux 1.2.0 [13] and Windows 95 were just hitting the shelves.

In recent years, we assisted to the rise of two distinct set of technologies that could, if refined, disturb the classical 3-tier architecture: (a) Javascript MVC framework and (b) NoSQL database. Indeed, Javascript frameworks are now able to consume HTTP APIs [14], [15] and construct dynamic page web and NoSQL databases provide HTTP APIs for client to consume [17], [18]. Figure 1 depicts a classical usage scenario for these technologies where Angular2 from Google (https://angular.io/) acts as the presentation layer, PHP 7 is operating as the application layer, and CouchDB is used for the data layer. First, the presentation layer sends an HTTP-POST request to the application layer for authenticating a user. The user email and password are passed as a form parameter of the HTTPS encrypted POST request. The application layer receives this request and sends another HTTPS encrypted POST request to the data layer. The second request is slightly different as we look for a user inside the database that matches the given email and password. If such user exists, then the data layer answers to the application layer with the user details and the application layers forwards this information to the application layer. In addition to the user details, the application layer creates an authentication cookie and sends it back the presentation layer in order to facilitate further requests.

As depicted by Figure 1, the responsibilities of the application layer in such technological environment are (a) proxying requests to the data layer API and (b) manage the authentication of users. In theory, one could build a CRUD (create, read, update, delete) two-tier application with a Javascript Framework and a NoSQL database. Especially considering that NoSQL database HTTP-API supports a map/reduce engine [19] that allow users to perform operations more complex that CRUD ones (i.e., sum, average, aggregation, filter, ...). Significant gaps exist, however, in these technologies for them to be able to remove the application layer of the three-tier architecture entirely: (a) read/write privileges and, (b) schema management. Indeed, NoSQL databases only provide privileges at the database level. Consequently, a user with the read (write) permission can read (write) all the database. Then, NoSQL databases are schema-less meaning that they do not have an enforceable definition of what should or should not be in the database. If the database API were to be accessible publicly, then, any malicious user could create database entries containing unexpectedly large documents resulting in a DOS (Denial Of Service) attacks.

As a result, companies using Javascript application for their presentation layer and NoSQL database for their data layer are developing and maintaining an application layer using a server-side language with the associated cost in workforce and servers.

To assess these gaps, we created an add-on for a popular

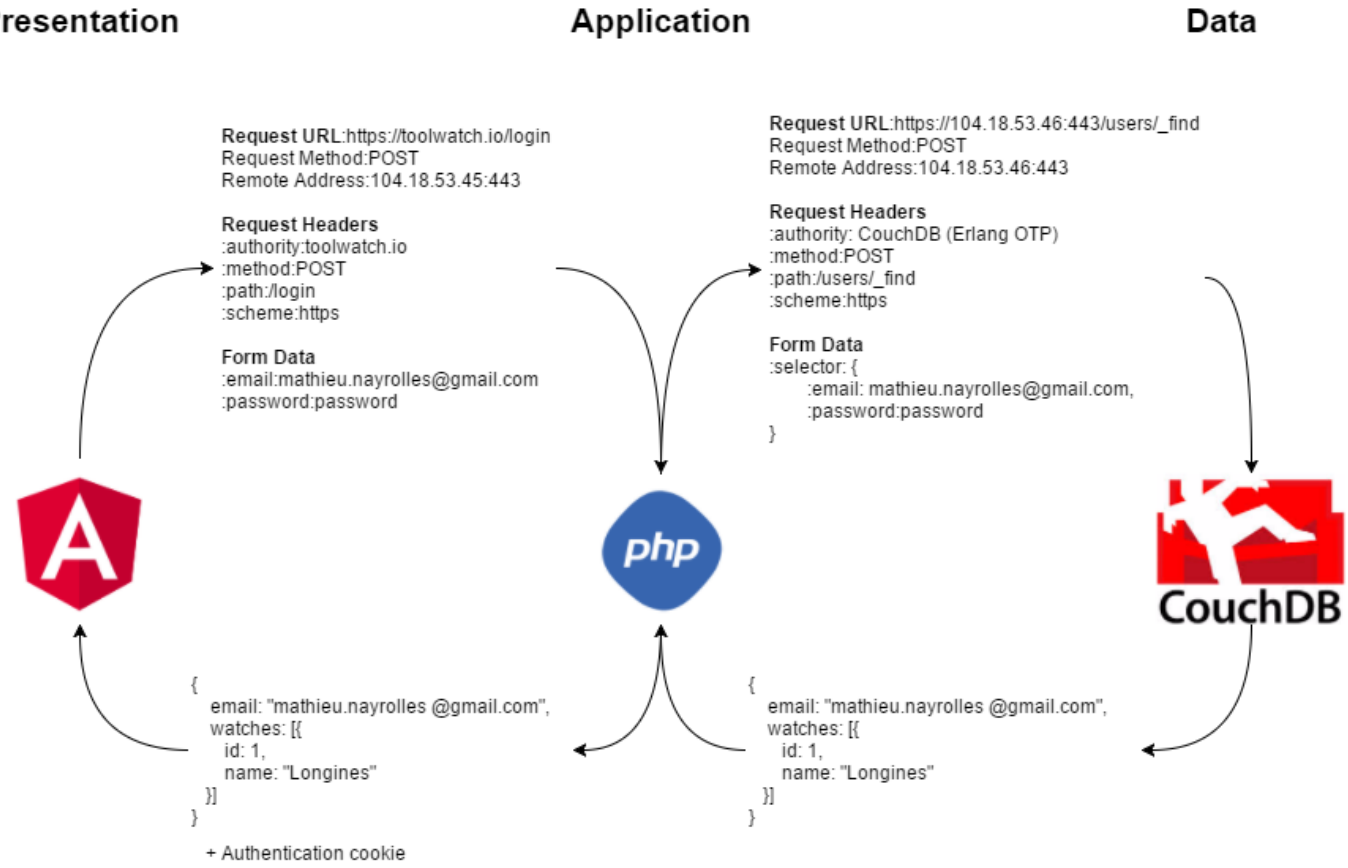

Fig. 1: Managing events happening on project tracking systems to extract defect-introducing commits and commits that provided the fixes

NoSQL database. More specifically, when a request is made to the NoSQL database through its HTTP-API, our add-on filters out unauthorized actions and ensures that the database access is not harmful. Also, our add-on provides missing functionalities such as access rights, account, and schema management using a JSON configuration file.

In this paper, we present how we overcame these limitations and created an efficient two-tiered architecture for web services. We experimented this new web-programming paradigm at Toolwatch; a web-based company that allows tens of thousands of clients to measure the accuracy of their mechanical watch every day, and found that operation costs can be reduced by 63% while improving performances by 34%. Also, the time required to develop new feature can be reduced by 41%.

The rest of this paper is organized as follows. Section III presents our approach while sections IV and V describe our case study setup and results. Then, section VI presents the threats to validity. Finally, sections II and VII present the related work and propose a conclusion to this paper.

II. Related Work & Background

In this section, we present the work that is related to ours. First, we elaborate on the different physical layouts that have been dominating web-based services in section II-A. Then, in section II-B we present some of the well-known programming models for web programming. These two particular areas: (a) physical architecture and (b) programming model for the web are related to our work as we propose a new way to use reactive and event-driven programming on a two-tiers architecture.

A. Physical Architecture

In Section I we presented layered architecture [1], [2]. More specifically, we introduced the three-tier or n-tier architecture which is an evolution of the older client/server architecture [20]–[22]. The three-tier architecture is a software architecture and a software design pattern, both [23]. It is composed, as its name suggests, of three-tier:

- Presentation tier: This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network.
- Application tier: The logic tier is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing.
- Data Tier: This tier consists of Database Servers. Here information is stored and retrieved. This tier keeps data

neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance.

The three-tier architecture is different from the MVC (Model-View-Controller) design pattern [24] as it requires that the application-tier never communicates directly with the data-tier.

In a web development context, the three-tier architecture refers to (a) A front-end web server serving static content, and potentially some are cached dynamic content. (b) A middle dynamic content processing and generation level application server, for example, Java EE, ASP.net, PHP platform. And (c), A back-end database, comprising both data sets and the database management system or RDBMS software that manages and provides access to the data.

In this paper, we propose to *revert* back to a two-tier architecture (also-known-as client-server) where we remove the application tier. The responsibilities of the application tier are transferred to the presentation and the data tier thus saving computational power, money and, workforce.

B. Programming Models for the Web

It exist plenty of programming models targeting web-development: Event-Driven Programming [25], [26], Reactive Programming [16], [27], Data-Centric Programming [28], [29] or Model-Driven Programming [30] to name a few. We consider our contribution to be part Event- Reactive-Driven and part Data-Centric as we use a reactive Javascript framework that communicates directly with a scalable, distributable data-hub (see Section IV). Consequently, in this section, we describe approaches belonging to the Event- Reactive-Driven and part Data-Driven Programming for web applications.

1) Event-Driven & Reactive Programming: Event-Driven or reactive programming aims to create software systems with high performance and resilience as argued by Dabek *et al.* [25]. It is a programming paradigm in which the program execution is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. Reactive programming build on that idea but with asynchronous data streams. Moreover, Dabek argues that events are a better means of managing I/O concurrency in server software than threads: events help avoid bugs caused by the unnecessary CPU concurrency introduced by threads. Elaine Cheong *et al.* proposed TinyGALS which is a globally asynchronous and locally synchronous model for programming event-driven embedded systems [26]. A more recent example of asynchronous I/O with an event-driven programming model is Node.js. Stefan Tilkov *et al.* explores why it performs better than multi-threading [31]. TinyGALS and Node.js rely on *mainstream* programming languages: Java and Javascript, respectively. However, the event-driven programming paradigm was also applied with dedicated language such as ESP* [32] or Flapjax [27]. On the one hand, ESP* was created behavior modeling of embedded system components and allows for (1) explicit states, (2) asynchronous events, and (3) conditional execution. On the other hand, Flapjax provides event streams, a uniform abstraction for communication within a program as well as with external Web services and the very core of the language is reactive.

Despite the renewed popularity of event-driven and reactive programming due to new JavaScript framework such as Angular, React or Backbone [14]–[16], these programming technics have been known for two decades. Elliot *et al.* introduced Fran (Functional Reactive Animation) in 1997. They were among the first to define behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. They use these new concepts for image animations.

Event-Driven & Reactive Programming using I/O concurrency manages to improve performances and resilience in the highly stressed system. However, these programming techniques also have drawbacks. Indeed, event-based programs are heavily composed of *callbacks*. This can make the program hard to understand, evolve and maintain as demonstrated by Fisher *et al.* [33]. Kambona *et al.* confirmed Fisher *et al.* experimentations by finding asynchronous spaghetti code that is difficult to maintain as the programs grow [34]. To ease these concerns, a different approach has been proposed such as asynchronous tasks [33], a collection of enforceable rules [35], guidelines for the underlying language itself [34] and integration with general imperative language [36].

2) Data-Centric Programming: Another, less popular, programming paradigm for web development is the data-centric paradigm inspired by the Datalog programming language. Datalog is a subset of Prolog mainly used as a query language.

The data-centric approaches for web development focus on providing programming methodology based on declarative networking [28], [37]. Declarative Networking is a programming methodology that enables developers to concisely specify network protocols and services, which are directly compiled to a dataflow framework that executes the specifications.

Another data-centric approach is to use Datalog to build distributed applications. Abiteboul *et al.* [29], [38] and Field *et al.* [35] proposed such approaches. One of their main contributions is a language that allows distributed data model where peers exchange messages as well as Datalog-style rules in a semantic manner. Grumbach *et al.* proposed something related: fixpoint semantics, which takes explicitly into account the in-node behavior and syntactic restrictions over the programs can ensure polynomial bounds on the complexity [39].

Alvaro *et al.* took another direction and built a data-centric approach for the web using the Overlog language to implement a "Big Data" analytics stack that is API-compatible with Hadoop and HDFS and provides comparable performance.

Our approach differs from previous works in a way that it does not strictly belong to one programming models but combines

two of them: event- reactive-driven and data-centric. Also, we improve on the data-driven programming model by bringing functionalities that once belonging the application tier to the data tier.

III. A Modern 2-Tier Architecture

In Section I we presented a classical usage scenario for the event- reactive-driven presentation layer, a classical application layer, and an HTTP-API enabled data layer. An HTTP-POST request is sent to the application layer for authenticating a user. The user email and password are passed as a form parameter of the HTTPS encrypted POST request. The application layer receives and forward the request to the data layer request and sends another HTTPS encrypted POST request to the data layer. Then, the response layer is transferred back to the presentation layer through the application layer. While we can envision programming paradigm that removes the application layer from the equation, we identified significant gaps: (a) read/write privileges and, (b) schema management.

In this section, we describe how we assess these three different shortcomings by building upon CouchDB (i.e., the data layer). These gaps could only be assessed in the data layer as the presentation layer, which is composed of HTML, CSS and, javascript; is shipped into the client computer. Consequently, we cannot enforce any policies as an ill-intentioned client could easily override our safeties net. The data layer, however, lies in a controlled environment and its code and policies are not accessible to the end-user.

A. Read/Write privileges

The first shortcoming to assess to remove the application layer and, thus, improve performance and resilience of web-based application is related to access rights. Every database, either relational or NOSQL, provide access rights management. However, to the best of our knowledge, major relational and NoSQL database manage such rights at table (collection) level. Consequently, each user can only have rights that span through the whole table (collection) and not only a few records of the table (collection). This is a problem as, for example, a user should only have access to his information and not the information of other users. Using only the mechanisms by major database vendors to manage rights cannot answer this problematic as we could only allow a new user to access the while user table (collection).

To asses this first challenge, we introduce the concept of ownership over records. In a NoSQL key-value store environment such as CouchDB (see Section IV), records are simple JSON documents. As an example, the following document display part of the actual Tool watches user with anonymized personal data:

```
{
 uuid:"6e1295ed6c29495e54cc05947f18c8af",
 name:"Jacques",
 lastname:"Dupond",
 pw_hash: "c7e3343645eb464c665d6ad905408271",
 email_preferences: { },
 login_locations: [],
 login_times: [],
 roles:["user"]
}
```

The user document is composed of seven fields. The first field, uuid, stands for Universally Unique IDentifier and is random identifier generated by the database. The uuid is used to uniquely identify any document in the database. We leverage the uuid of users to create ownership and enforce appropriate read/write rights over the document. The `roles` array defines the roles attributed to this user. Consequently, a watch document with owned by Jacques-Dupond would follow the following structure. As a reminder, Toolwatch.io is a web-based platform allowing users form to measure the drift of their mechanical watches.

```
{
 uuid:"85fb71bf700c17267fef77535820e371",
 name:"Submariner",
 brand:"Rolex",
 measures:[
     {
         measureRefTime:1427627730,
         userRefTime:1427627739,
         accuracyRefTime:1427718340,
         userRefTime:1427718356,
         accuracy:6.67475996
     },
     ...
 ]
 owner:"6e1295ed6c29495e54cc05947f18c8af"
}
```

In addition to the fields required by Toolwatch.io to compute the accuracy of a given watch, the watch document contains the `owner` field. The owner field acts as a flag that indicates which users owns this particular record in a similar way a foreign key would in a relational database. Despite the fact that implementing a *foreign key system* on a NoSQL database, which has been conceptualizing to operate without relational dependencies between element and another schema can seems counter-intuitive, in sections IV and V, we demonstrate the pertinence, in terms of performances and code comprehension of such a *infringement* to the NoSQL way of thinking.

To enforce the reading and writing rights over documents we created a two-fold process. First, we define user roles and permissions into a JSON file that our database engine loads at startup.

```
{
     [
         name:"user",
```

```
        rights:["owns"],
    ]
}
```

The second step of enforcing these permissions policies is to modify, on the fly, every incoming request with the according to the user rights. As an example, the after login request from Section I, the follow-up request is to fetch the watches of the user.

```
Request URL:
https://104.18.53.46:443/watches/_find
Request Method:POST
Status Code:200
Remote Address:104.18.53.46:443

Request Headers
:authority: CouchDB (Erlang OTP)
:method:POST
:path:/login
:scheme:https
:csrf:b00b8d275731ead07a8b2df9580dfa88

Form Data
:selector: { }
```

The `selector`—which is the NoSQL equivalent of the RDBMS `WHERE` is empty. On the fly, we modify the `selector` content to match the rights belonging to the user.

```
Request URL:
https://104.18.53.46:443/watches/_find
Request Method:POST
Status Code:200
Remote Address:104.18.53.46:443

Request Headers
:authority: CouchDB (Erlang OTP)
:method:POST
:path:/login
:scheme:https
:token:b00b8d275731ead07a8b2df9580dfa88

Form Data
:selector: {
    owner: "6e1295ed6c29495e54cc05947f18c8af"
}
```

We can identify which user the request originated from using the `token` header. Each time a user makes a request, he receives a new token to use for its next request. This is a well-known technique in the prevention of CSRF or XSRF (Cross-site request forgery) attacks [ref] that is used in parallel of classical cookie based session. The point being that cookie can be stolen and used on different machines to *stole* user identity without actually knowing the credentials of the target. It is also known as session hijacking [ref].

Using the same mechanisms, we can define more complex roles such as the `shop` role. A user with the `shop` role can manage the information of its customers. As an example, a `shop` user:

```
{
 uuid:"dcc2ea988aefd77bd92590b849f6f91",
 name:"shop_owner",
 lastname:"shop_owner",
 pw_hash: "933edde662cb6021151c2b728d00876e",
 customers:["85fb71bf700c17267fef77535820e371"]
 email_preferences: { },
 login_locations: [],
 login_times: [],
 roles:["shop"]
}
```

and the rights:

```
    [
        name:"shop",
        rights:["owns", "customers.owns"]
    ]
```

In such a case, a request with an empty `selector` made by a `shop` user will be, on the fly, agreement with the following conditions:

```
Request URL:
https://104.18.53.46:443/watches/_find
Request Method:POST
Status Code:200
Remote Address:104.18.53.46:443

Request Headers
:authority: CouchDB (Erlang OTP)
:method:POST
:path:/login
:scheme:https
:token:d6741b7ac2a0299686bfa0dfa5811acc91473873952

Form Data
:selector: {
    $or:{
        owner: "dcc2ea988aefd77bd92590b849f6f91",
        owner: $in{["85fb71bf700c17267fef77535820e
    }
}
```

The `$or` selector allows to have a boolean or between two statements and the `$in` selector returns true if the field values matches one of the specified values. The behaviors of the `$or` and `$in` operators are comparable to the SQL `WHERE X or Y` and `WHERE IN`.

The actual implementation of the on-the-fly modification of data incoming to the data tier is a simple C++ program that scans and preempts incoming HTTP requests. The modification itself is handle by the Microsoft C++ JSON framework Casablanca [ref]. Once the request is preempted, the JSON is analyzed and compared to the rights of the user making the request and modified if needs are.

B. Schema management

NoSQL databases are known to operate without a schema. It means that documents do not have to obey a particular pre-set structure. Indeed, any one record can be different in the same collection. We displayed this particularity in the previous section while presenting two different Toolwatch users. The first user is called a *standard* user at Toolwatch. This type of users operates as a single entity and have rights over its resources. The second type of users is *shop* users. The *shop* user has the possibilities to manage the resources of users that, for example, bought their timepieces in their shop. Their records are similar with the exception of the customer's field:

```
customers:["85fb71bf700c17267fef77535820e371"]
```

that *shop* users have and *standard* user do not.

This NoSQL particularity provides flexibility as the information stored in the database can evolve over time without a need to modify a schema and make record obey this schema.

In the model we propose, this open the door to a security flaw known as server-side denial of service attacks [ref]. Classical denial of service attacks (DDOS) involves flooding a particular server with a multitude of request incoming from tens of thousand of machines for a prolonged period in the hope that the application would not be able to handle this unexpected load and crashes. At the very least, the quality of service of the targeted application is impacted as it takes longer for legitimate customers to be served. The server-side denial of service is variant that makes the server perform a time-consuming operation and hinder its ability to serve the request. The classical example of server-side denial of service attacks is to use to use very long password when creating your account. Indeed, the password will be hashed, and hashing is a costly operation. If your application layer does not limit the length of submitted information, one can submit a 5 Mb long string and increase significantly the server CPUs' usage.

In the framework of this study, a server-side denial of service would be relatively easy to perform as any one field of any record is not checked for content-type or length. Consequently, an ill-intentioned user could simply submit documents containing millions of fields, each field composed of very-long string. The aftermath will be two-fold. First, the database engine will have to process and index all this information and, second, the bogus information will occupy a significant part of the available storage. The quality of service of the application could degrade down to 0% either because of lack of computational power or lack of disk space.

To assess these potentials threats, we define a schema for the database using a JSON file. This schema is loaded by our request interceptor when the server starts. Each intercepts requests undergo a thorough validation of the schema before applying the rights described in the previous section. If a request does not comply with the schema, then, the request is discarded and do not reach the database engine.

```
{
 user:{
         uuid:{type:"string", limit:256},
         name:{type:"string", limit:256},
         lastname:{type:"string", limit:256},
         pw_hash: {type:"string", limit:256},
 },
 ...
}
```

The previous figure presents part of the schema definition for Toolwatch's users. Each collection is composed of different field, and each field is typed and limited in length. When incoming requests are intercepted as described in the previous section, we verify that the information submitted with the request comply with the schema. Consequently, the database engine only receives requests that are schema-compliant and, therefore, alleviate the risk of server-side denial of service.

To summarize our approach, we remove the application tier of the three-tier (or n-tier) architecture and splits its responsibilities between the presentation and the data side. Javascript Framework on the presentation sides is able to generate dynamic web-pages by consuming HTTP-API while being served to the client statically. This makes up for the capacity of the application side to generate dynamic HTML pages and serve them to the client. The other responsibility of the application side is to communicate with the data tier. Modern NoSQL databases provide HTTP-API that can be consumed from a remote location. These data-side HTTP-API, however, do not provide adapted right management nor descent security protection. Consequently, this data-side HTTP-API are not exposed to end-users. We propose a simple HTTP request inceptor that intercepts requests before they hit the database engine. Our interceptor applies rights and security checks to the request and its contained information to fill the gap of current data-side HTTP-API: (a) right and (b) schema management.

IV. Case Study Setup

In this section, we present the setup of our case study regarding usage scenario, comparison process, and evaluation measures.

To assess the effectiveness of our programming model we evaluate regarding performances against *classical* programming models while performing a set of real-life scenarios.

We evaluate the performances by comparing the performances of the same use case while using our programming model, PHP, Java, Ruby on Rails. The choice of these particular programming languages for evaluation were motivated by the fact that they are different yet popular in the web-development community [ref]. First of all, Php and Ruby have interpreted languages while Java is compiled into an intermediate language that is later executed by the JVM (Java Virtual Machine). The time to process a request is divided into several blocks:

- Queuing: The time before the browser sends the request.
- DNS Lookup: The time to resolve the top level domain name into an address IP
- Initial connection: The time required to establish a TCP/IP connection with the remote server
- SSL: The time required to establish an encrypted connection with the remote server.
- Sending Time: The time to send the request.
- Time To First Byte (TTFB): The time the browser wait for the first byte of the response.
- Content Download: The type it took to download all the bytes of the response.

Some of this time blocks are not dependent on the programming model used. Indeed, queuing, DNS Lookup, Initial connection, SSL, sending time and content download depends on network speed and congestion. For this reason, we only use the TTFB value, which is the actual time it took to compute a response and start sending it back to the user, to compare implementation relying on a classical application layer and our programming model.

We tested each scenario ten times with 1, 100,1000 and 10,000 concurrent users for each implementation. The results are then fed to a boxplot. The boxplot is a statical method that clusters the values into groups. In the next section, we report the value for the 1st quartile, the median and the 3rd quartile (lower response TTFB is better). This gives an accurate representation on how much time the system needs to perform a scenario and how each implementation scales.

1) User creation: In this scenario, a new user creates an account and receives confirmation.

2) Resources Creation: In this scenario, users create new resources (i.e., a watch and an associated measure in the toolwatch context).

3) Resources Retrieval: In this scenario, the user retrieves the created resources that it owns and tries to assess resources owned by other users.

4) Resources Updates: In this scenario, user updates the retrieved resources that it owns and tries to update resources owned by other users.

5) Resources Deletes: In this scenario, user updates the retrieved resources that it owns and tries to assess resources owned by other users.

Note that, in our experimentations the application layer container and the data layer container for Php, Java and RoR were on the same physical machine. Consequently, the reporter times do not account for network time between the data layer and the application layer. The only network time requires the time it takes establish a connection to the same machine on a different port and negligible transfer time. This said, actual network transfer between the application layer is not negligible on real world application. At Toolwatch, the average time it took (before the deployment of the programming model we propose) was averaging at 345 ms with an application layer located in a data center in Lausanne (Switzerland) and a data layer located in Roubaix (France).

In the same manner that we did not report the time it rake for DNS resolution and content downloading for the communication between the presentation layer and the application layer, we did not penalize the three-tier architectures by measuring the network time between their application and data layer. It is of opinion that it will not be fair ad this largely depends on network topologies (i.e., are the application and data layer on the machine, data center, country ?).

It is obvious that the programming model proposed in this paper overcomes three tiers ones as we have one less network operation to handle (i.e., from the application to the data layer).

We ran the test on an Intel I7 3.8 GHz with 8Ghz RAM Linux computer running Docker container. Docker is an implementation of the LLXC container for Linux that is a form of visualization. We had one container with the application layer and one container for the data layer. The presentation layer was simulated by the Apache benchmark library that can craft HTTP request and simulate concurrent users.

A. Operations Cost

Regarding the evaluation of the operations costs of using our programming model uses descriptive statistics computed on scale down operated at Toolwatch. For confidentiality reasons, we cannot disclose the actual numbers. However, descriptive statistics on some servers spared and operations costs reflect the benefits of our contribution.

B. Program Comprehension, Maintenance et Evolution

Finally, concerning the program comprehension, maintenance et evolution we report descriptive statistics on the number of lines of code per tier before and after the migration to the new system. LoC and NoC cannot be used as a direct indicator of code maintainability, ease of evolution and comprehension (i.e. a short program hard to understand). It is our opinion. However, that is a significant reduction of LOC and NOC with equal functionalities is a desirable effect.

V. Case Study Results

TODO: Plot the results

VI. Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to introduce new programming paradigm. It is possible that the selected use-cases share common properties that we are not aware of and therefore, invalidate our results. However, the tested use-cases are the *classical* ones for web applications (i.e., Login, Creating/Updating/Deleting resources). Also, we see a threat to validity that stems from the fact that we conduct this study on an industrial system. While unlikely, The results may not be generalizable to open source systems. We intend to undertake these studies in future work.

The programs we used in this study are based on the Javascript (presentation layer) and Erlang programming language (data layer). This can limit the generalization of the results to projects written in other languages. While it exists other open-source NoSQL databases providing HTTP API that could be improved according to our approach, to the best of our knowledge, every front-end framework using event- reactive-driven programming is based on Javascript.

In conclusion, internal and external validity have both been minimized by choosing a set of *classical* uses cases, using functionalities available in many programming language and framework.

VII. Conclusion

In this paper, we presented a new web-programming model that combines event- reactive driven and data-centric models. Our programming model intercepts send to the data tier and modify them, on the fly, to enforce rights and provide security effectively replacing the applications tier. This model has the particularity to be deployed on a 2 tier architecture while yielding the same beneficial effects as 3 tier architecture: scalability and resilience. Also, web application build with our model proved to be faster, smaller and cheaper to operate than classical applications.

As future work, we plan to experiment with other font end frameworks and databases to make our implementation cross-product. Finally, we plan on open sourcing our implementation for the community to use as soon as ToolWatch specifics are removed from the source base.

References

[1] W. W. Eckerson, "Three tier client/server architectures: achieving scalability, performance, and efficiency in client/server applications," *Open Information Systems*, vol. 3, no. 20, pp. 46–50, 1995.

[2] R. Hirschfeld, "Three-tier distribution architecture," *Pattern Languages of Programs (PloP)*, 1996.

[3] J. C. Leite, D. M. Kusic, D. Mossé, and L. Bertini, "Stochastic approximation control of power and tardiness in a three-tier web-hosting cluster," in *Proceeding of the 7th international conference on autonomic computing - icac '10*, 2010, p. 41.

[4] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *2010 11th ieee/acm international conference on grid computing*, 2010, pp. 41–48.

[5] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long, "Optimal Cloud Resource Auto-Scaling for Web Applications," in *2013 13th ieee/acm international symposium on cluster, cloud, and grid computing*, 2013, pp. 58–65.

[6] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling Web Applications in Heterogeneous Cloud Infrastructures," in *2014 ieee international conference on cloud engineering*, 2014, pp. 195–204.

[7] R. Prodan and S. Ostermann, "A survey and taxonomy of infrastructure as a service and web hosting cloud providers," in *2009 10th ieee/acm international conference on grid computing*, 2009, pp. 17–25.

[8] M. Zhou, R. Zhang, D. Zeng, and W. Qian, "Services in the Cloud Computing era: A survey," in *2010 4th international universal communication symposium*, 2010, pp. 40–46.

[9] I. Peña-López and Others, "Manual for measuring ICT access and use by households and individuals," 2009.

[10] P. Windrum, "Leveraging technological externalities in complex technologies: Microsoft's exploitation of standards in the browser wars," *Research Policy*, vol. 33, no. 3, pp. 385–394, 2004.

[11] B. Eich, Brendan, Eich, and Brendan, "JavaScript at ten years," in *Proceedings of the tenth acm sigplan international conference on functional programming - icfp '05*, 2005, vol. 40, p. 129.

[12] B. A. Myers and B. A., "A brief history of human-computer interaction technology," *interactions*, vol. 5, no. 2, pp. 44–54, Mar. 1998.

[13] D. Bretthauer, "Open source software: A history," *Information Technology and Libraries*, vol. 21, no. 1, p. 3, 2002.

[14] M. Hevery and A. Abrons, "Declarative web-applications without server," in *Proceeding of the 24th acm sigplan conference companion on object oriented programming systems languages and applications - oopsla '09*, 2009, p. 801.

[15] M. Wilson, *Building Node Applications with MongoDB and Backbone*. " O'Reilly Media, Inc.", 2012.

[16] C. Gackenheimer, "What Is React?" in *Introduction to react*, Springer, 2015, pp. 1–20.

[17] R. Cattell and Rick, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, p. 12, May 2011.

[18] R. P. Padhy, M. R. Patra, and S. C. Satapathy, "RDBMS to NoSQL: reviewing some next-generation non-relational

database's," *International Journal of Advanced Engineering Science and Technologies*, vol. 11, no. 1, pp. 15–30, 2011.

[19] J. Dean and S. Ghemawat, "MapReduce," *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan. 2008.

[20] M. J. D'Amore and D. J. Oberst, "Microcomputers and mainframes," in *Proceedings of the 11th annual acm siguccs conference on user services - siguccs '83*, 1983, pp. 7–17.

[21] J. E. Israel, J. G. Mitchell, and H. E. Sturgis, "Separating data from function in a distributed file system," Xerox, Palo Alto Research Center, 1978.

[22] D. K. Gifford and D. K., "Weighted voting for replicated data," in *Proceedings of the seventh symposium on operating systems principles - sosp '79*, 1979, pp. 150–162.

[23] Microsoft, "Deployment Patterns (Microsoft Enterprise Architecture, Patterns, and Practices).".

[24] R. E. Johnson, "Documenting frameworks using patterns," *conference proceedings on Object-oriented programming systems, languages, and applications - OOPSLA '92*, pp. 63–76, 1992.

[25] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th workshop on acm sigops european workshop: Beyond the pc - ew10*, 2002, p. 186.

[26] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "TinyGALS," in *Proceedings of the 2003 acm symposium on applied computing - sac '03*, 2003, p. 698.

[27] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a programming language for Ajax applications," in *Proceeding of the 24th acm sigplan conference on object oriented programming systems languages and applications - oopsla 09*, 2009, vol. 44, p. 1.

[28] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking," *Communications of the ACM*, vol. 52, no. 11, p. 87, Nov. 2009.

[29] S. Abiteboul, M. Bienvenu, A. Galland, and É. Antoine, "A rule-based language for web data management," in *Proceedings of the 30th symposium on principles of database systems of data - pods '11*, 2011, p. 293.

[30] S. J. Mellor, M. Balcer, and I. Foreword By-Jacoboson, *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[31] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, Nov. 2010.

[32] V. C. Sreedhar and M.-C. Marinescu, "From statecharts to ESP," in *Proceedings of the 5th acm international conference on embedded software - emsoft '05*, 2005, p. 48.

[33] J. Fischer, R. Majumdar, and T. Millstein, "Tasks: language support for event-driven programming," in *Proceedings of the 2007 acm sigplan symposium on partial evaluation and semantics-based program manipulation - pepm '07*, 2007, p. 134.

[34] K. Kambona, E. G. Boix, and W. De Meuter, "An evaluation of reactive programming and promises for structuring collaborative web applications," in *Proceedings of the 7th workshop on dynamic languages and applications - dyla '13*, 2013, pp. 1–9.

[35] J. Field, M.-C. Marinescu, and C. Stefansen, "Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications," *Theoretical Computer Science*, vol. 410, no. 2, pp. 168–201, 2009.

[36] C. Schuster and C. Flanagan, "Reactive programming with reactive variables," in *Companion proceedings of the 15th international conference on modularity - modularity companion 2016*, 2016, pp. 29–33.

[37] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, I. Stoica, B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *Proceedings of the twentieth acm symposium on operating systems principles - sosp '05*, 2005, vol. 39, p. 75.

[38] S. Abiteboul, O. Benjelloun, and T. Milo, "Positive active XML," in *Proceedings of the twenty-third acm sigmod-sigact-sigart symposium on principles of database systems - pods '04*, 2004, p. 35.

[39] S. Grumbach and F. Wang, "Netlog, a Rule-Based Language for Distributed Programming," Springer Berlin Heidelberg, 2010, pp. 88–103.