Composing Internet of Things Platforms in Smart Grid

Ervin Varga, *Bojan* Blagojević, and *Dejan* Mijić

Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, Novi Sad, Serbia

Abstract. The abundance of smart appliances and concomitant data in Internet of Things (IoT) poses new industrial challenges. Devices must be managed, and data efficiently harvested. IoT platforms are the rescue; they are key architectural components in combatting complexity and scalability issues. This paper proposes a pragmatic composition method of these platforms, hence helps solving IoT interoperability conundrums. The described approach results in a higher-level abstraction, that shields user applications from the underlying turmoil. Our article presents an approach of wrapping such a mesh under a high-level domain API. To verify the soundness of our proposal we have implemented a fully functional proof of concept implementation. The source code is freely available at request. The outcome decisively demonstrates the proposed method's feasibility, power, and usefulness.

1 Introduction

The vast amount of data, produced by sensors and smart appliances, hides many opportunities to boost both business and operational intelligence [1]. For example, in manufacturing there is an urgent need to predict machine failures, and schedule maintenance work accordingly. The return on investment in applying IoT is obvious; less downtime results in huge savings. Moreover, by creating historical reports, the management may better organize supply chains, and make sound business plans [2]. Before the IoT era, the industry relied on traditional SCADA systems, or similar centralized solutions, for factory automation. Such approaches had serious limitations stemming from data compartmentalization. IoT tries to dynamically bundle disparate data channels, as multiple data sources may provide new insights (something not possible by looking at each of them separately). Nonetheless, IoT also introduces novel challenges. The major obstacles toward making IoT a commodity can be segregated into the following categories: device management, real-time event handling, data analytics (embraces Big Data problems) [3], end user applications, and methods to develop IoT systems [4]. Security is a cross-cutting concern that is exaggerated by a higher exposure of devices to all sorts of attacks i.e., the attack surface is volatile and voluminous [5, 6]. However, the enormous gain from harvesting IoT's capabilities mandates us to find resolutions to the previously enrolled issues.

An IoT platform is a kind of an enabler such as a powerful software framework and integrated development environment for software development (nobody is writing million lines of code in assembly, nor is it a conceivable feast). At one point, it is impossible to manually tackle the task of controlling myriad of devices that swamp the target system with data. This is where an IoT platform plays a central role. It consists of components to manage devices, incorporates an event handling engine, and has facilities to perform data analytics. The large number of both commercial and open-source offerings (like Amazon Web Services IoT, Microsoft Azure IoT, Mainflux, Eclipse IoT, Kaa) is an excellent proxy for showcasing the importance of an IoT platform.

End users rarely embark on extensive software development; they prefer appropriate applications and services. Although, these could be delivered in many shapes, their common goal is to implement the required features. The same is true in the case of IoT. Users wouldn't use IoT platforms directly, but search for custom applications and services that will expose suitable, high-level Application Programming Interfaces (API) or Graphical User Interfaces (GUI). Figure 1 shows the major use cases pertaining to customers. These are expected to be delivered in the form of UI dashboards, Web service APIs (SOAP or REST), and

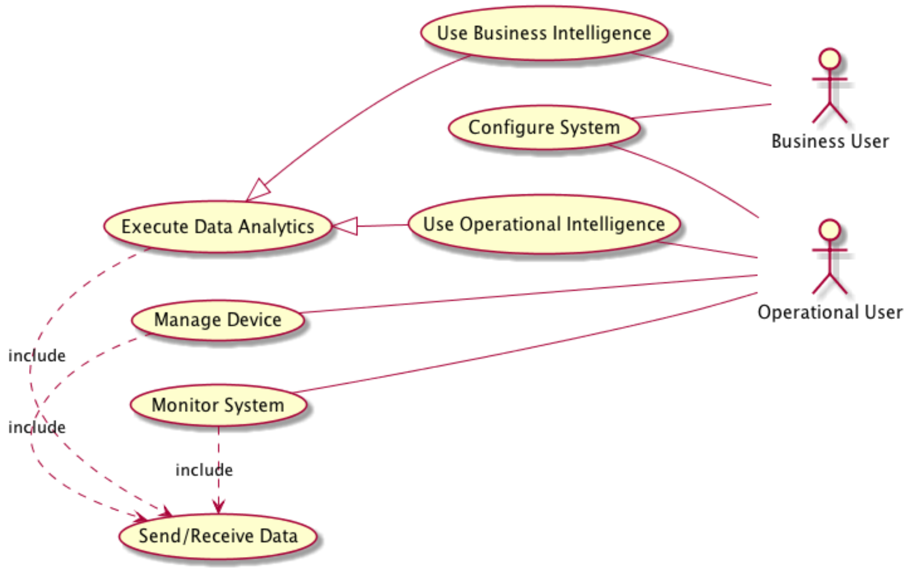

Fig. 1 The major IoT use cases, and their relations to the two groups of users: business and operations.

straightforward configuration mechanisms. In other words, the inner details of an IoT platform should be properly hidden.

The large number of data sources cannot be efficiently encompassed by a single IoT platform (either on premise or in the cloud). Consequently, each IoT platform is responsible only for a subset of devices; IoT platforms are frequently organized into a mesh network, having devices as leafs. The wisdom is based on the main tenet of edge computing, i.e., bring processing near data. This has multiple benefits, as listed below:

- Latency is reduced when handling critical events (geographical scalability).
- Irrelevant events are filtered out, thus only crucial ones are passed further (load scalability)
- Data handling logic is distributed; hence transformations happen early (functional scalability)

Evidently, IoT platforms must interoperate to form such networks, and make application development easier. This problem has been partially tackled by the EdgeX Foundry open-source umbrella project (see https://www.edgexfoundry.org), which aims to standardize what happens between devices and IoT platforms. Such a standardization prevents vendor lock-in at device level, thus enables consumption of data from disparate sources (many times combining data sources happen in an ad-hoc fashion). The authors are part of the core team of the Mainflux open-source IoT platform (https://www.mainflux.com), and members of the EdgeX Foundry project.

This paper describes a backend application/service construction method, that provides a domain-specific high-level API, and allows single point of control (administrative scalability) for users. The work is accompanied by the proof of concept (POC) implementation, in the domain of smart grids (a principal territory of IoT [7]).

2 Method

Our IoT application/service construction method embodies the next three characteristics, which are elaborated in separate sub-sections:

- API-driven development [8]
- Micro-services architectural style [9, 10]
- Stratified (layered) architectural pattern [10]

2.1 API-Driven Development

Figure 2 shows the UML activity diagram depicting the metaprocess of creating an API. A real (concrete) process would be an instance of this metaprocess configured with a set of principles, patterns, and techniques applied at each activity [8]. For example, in case of a RESTful API, there are additional constraints and specifics how to properly craft such an API [8, 11, 12], and in what ways is the concrete API process different than in the case of a SOAP based API [13].

Here are the most salient benefits of starting with an underpinning use case [8]:

- It encompasses behavioral requirements.
- It has a clear goal from the primary actor's perspective.
- It has a designated level helping to identify the target software layer, where this use case belongs.
- It lists all the stakeholders interested in the use case.
- It describes the main scenario, possibly with alternate flows.

The use case must be validated to properly reflect the intention of the primary actor. The use case later serves as a reference to verify the API. A use case is the generator of abstractions, which will become part of an API. Note that a single use case could entail multiple APIs if someone decides, for example, to apply the interface segregation principle to enhance maintainability, usability and reusability (this is reflected in the grouping abstractions meta-activity on Figure 2) [8].

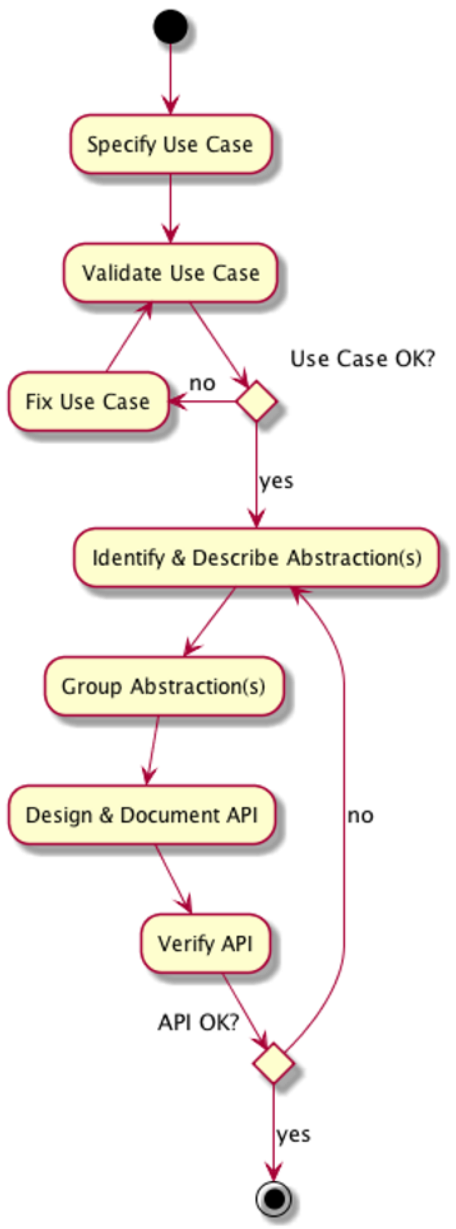

Fig. 2. The use-case-driven API metaprocess, where use cases play a central role in shaping the seams of the system's architecture (adapted with permission from [8]).

Figure 3 shows a typical build-up of an IoT platform (in this case Mainflux), that would serve as a model to explain the importance of use cases, and how they impact APIs. On the left side, we see the gateway facing devices, while on the right, we observe an integration point with custom applications. Devices and applications are on a completely different scale and abstraction level. Consequently, the associated APIs would serve a different purpose. Device APIs reflect specific low-level communication protocols (like, Level 1 REST, MQTT, CoAP, OSLP, Modbus, etc. [2, 7]) and goals; custom applications demand higher-level resource APIs (usually level 2 REST according to the Richardson's maturity model [12]), that hide device handling complexities, and provide aggregated views on the system.

On the other hand, custom applications would realize even more domain specific APIs for end users. These would be truly RESTful (level 3 REST), completely shielding users from the underlying IoT platforms. This approach is compliant with the tenet of the layered design, where as we move up on the layers' ladder we encounter more business oriented abstractions.

APIs also help in organizing work across distributed development teams, that are often geographically spread out with concomitant communication issues, which is crucial at the scale of IoT. A mobile application should be developed independently from an extension to the core IoT platform, or a new firmware sitting on devices. Finally, domain specific vertical APIs boost productivity of custom mobile applications, since working against IoT platform (or devices) would be cumbersome, and require a steep learning curve.

2.2 Micro-Services Architectural Style and the Layered Architectural Pattern

The micro-services architectural style is inherently based upon the layered design, and revolves around the following principles [9]:

- Domain-driven design [14]
- High automation with autonomous deployment model (aligned with the continuous delivery paradigm)
- Information hiding and encapsulation
- Decentralization of functional aspects of a system
- Dependability via failure isolation and powerful observability

Figure 4 depicts the module decomposition view of a generic vertical micro-service exposing a RESTful API toward clients (these may be mobile applications providing a decent user interface for users). The service communicates via proxies with the corresponding IoT platforms. Nonetheless, the existence of these platforms can be either completely hidden, or published in a very abstract manner. For example, the platform may be mentioned as a data source to be grouped with other similar units to form a composite. At any rate, none of the intricacies of dealing with platform specific details should be revealed in the service's API.

It is easy to notice layers inside the micro-service: API provider, engine, and database (frequently a relational database management system for the sophisticated query mechanism). The HTTP interface is common for all REST levels (from 1 to 3). The IoT platforms are treated as black boxes (they belong to a lower layer). Nevertheless, it is possible to peak into one of them to unravel a separate world of micro-services. This is illustrated on Figure 3 in the case of the Mainflux platform. The micro-services there are used

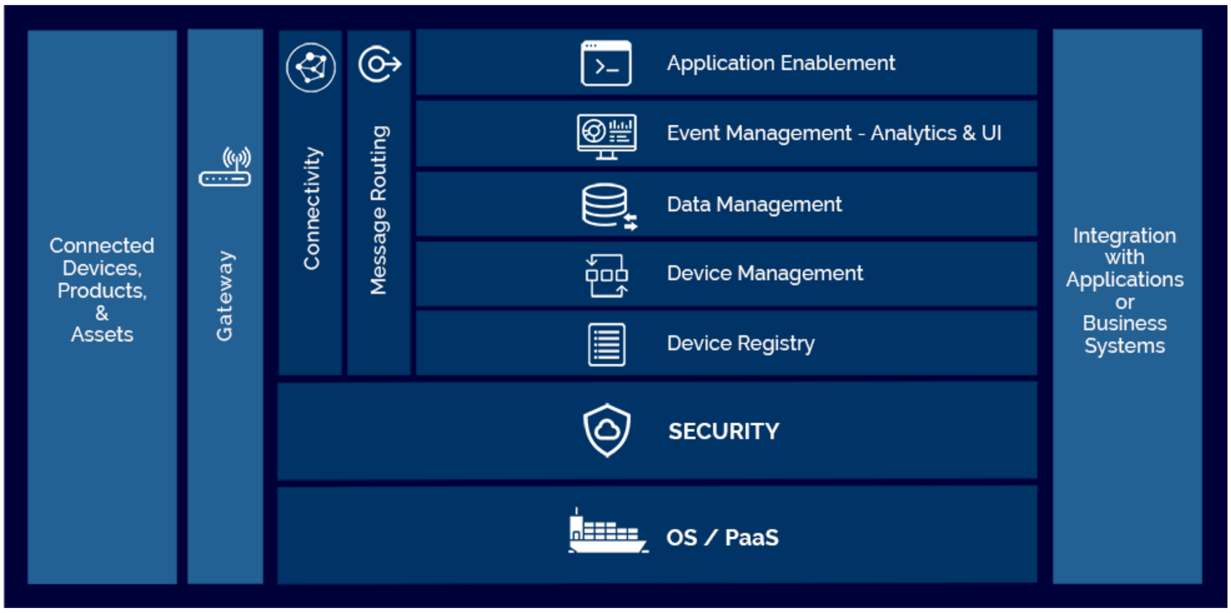

Fig. 3. The Mainflux software infrastructure stack, that contains all necessary components and micro-services for implementing fully functional IoT solutions (adapted with permission from http://mainflux.com). PaaS stands for Platform as a Service.

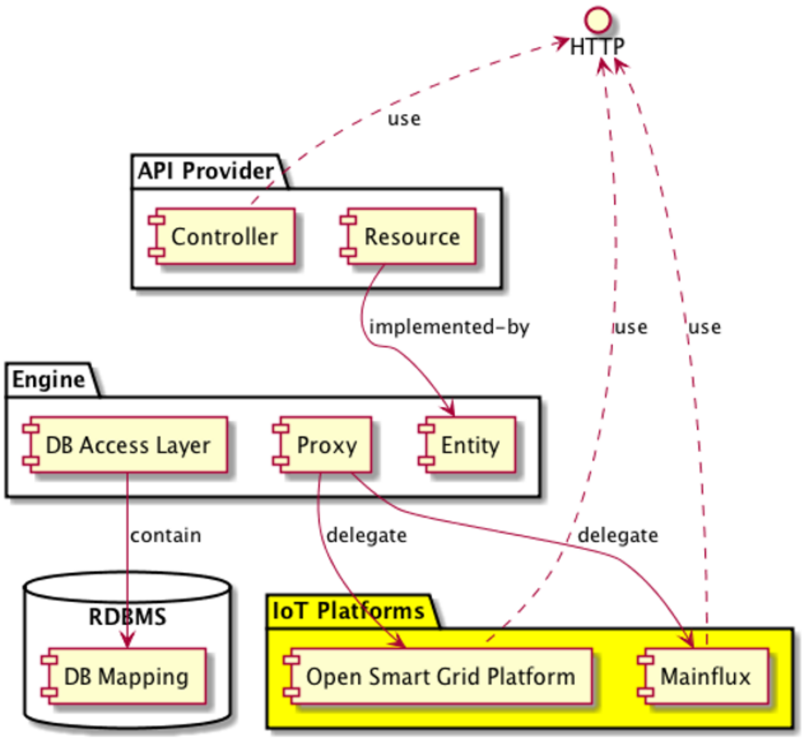

Fig. 4. The universal structure of a vertical IoT micro-service.

to implement the IoT platform in a scalable and fault-tolerant fashion. This is the principal power behind a layered design; it is always possible to zoom in/out as necessary (abstractions are major design artifacts to tame complexity, and layers group them in the most natural manner).

3 Result

The POC illustrates the method from the previous section. It showcases the management of microgrids as major building blocks of a smart grid. Each microgrid is controlled by a separate IoT platform, which is the Open Smart Grid Platform (OSGP). The POC uses the OSGP's micrograids domain for controlling and monitoring micrograids. Our custom application partially realizes the vision of OSGP's load management (https://opensmartgridplatform.org/load-management/). The POC offers a new OSGP domain to control the electricity load.

Besides getting loads from microgrids, the POC expands the load management use case with microgrids administration API. It allows an end user mobile application to create and delete microgrids in the system, group them together, and subscribe for reports about aggregated loads. The load management API is specified using on Open API Specification v3.0 (https://www.openapis.org/) leveraging JSON API v1.0 in the background (http://jsonapi.org/). The service is implemented as a micro-service in the Scala programming language using similar layering as depicted on Figure 4. Deployment is done via Docker Compose (see https://www.docker.com).

The end-points of the POC's Resource API are classified into three groups: microgrids, groups, and subscribers. The first denotes operations to manage microgrids, the second represents operations to handle groups, and the last contains end-points for dealing with subscriptions. This delineation of the API is the application of the interface segregation principle [15] in the realm of RESTful services.

The management interface would be driven by a separate client application. By using a callback notification mechanism (via the subscription facility), the communication patterns between the POC and each IoT platform may vary independently of subscribed clients; they are completely shielded from these details. Also, the POC's API is on a considerably higher level, than those used for application integration purposes in IoT platforms. An elevated domain specific API is crucial to enable massive adoption of IoT, and make the benefits available to end users.

4 Discussion

To expand the number of use cases related to IoT we need to attain the following objectives:

- Use IoT platforms as basic building blocks.
- Standardize the edge to allow interoperability among devices and IoT platforms.
- Combine IoT platforms under a domain specific umbrella API.

This paper shows a generic method to accomplish the last goal from the above list. The method is testified by a full POC system, that is freely available on-line. Once there is a high-level API it may be used by various user friendly mobile applications. Moreover, the API itself may be treated as a key asset, and published via API management platforms (like, Apigee, 3scale, etc.). According to [16] around 45-80% of market spending in IoT will be targeted for applications and value-added services. Having sophisticated vertical APIs is the precondition to achieve this forethought.

Our future work is related to incorporate security mechanism into the POC, and integrate many more different IoT platforms. Also, we plan to extend the load management related functions to support the establishment of dynamic pricing model at the level of microgrids. Such attempts are buttressed by the advancements in blockchain technologies for local electricity markets [17].

References

1. K. Lakhani and M. Iansiti, The Internet of Things, Harvard Business Review, 2014.
2. R. Barton, G. Salgueiro, and D. Hanes, IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things, Cisco Press, 2017.
3. H. Geng, Internet of Things and Data Analytics Handbook, John Wiley & Sons, 2017.
4. I. Jacobson, I. Spence, and P.W. Ng, "Is There a Single Method for the Internet of Things?", Communications of the ACM, Vol. 60, No. 11, Pages 46-53, 2017.
5. N. Dhanjani, Abusing the Internet of Things, O'Reilly, 2015.
6. S. Smith, The Internet of Risky Things, O'Reilly, 2017.
7. O. Elloumi, D. Boswarthick, and O. Hersent, The Internet of Things: Key Applications and Protocols, John Wiley & Sons, 2012.
8. E. Varga, Creating Maintainable APIs - A Practical, Case-Study Approach, Apress, 2016.
9. S. Newman, Building Microservices, O'Reilly, 2015.
10. M. Richards, Software Architecture Patterns, O'Reilly, 2015.
11. R. Daigneau, Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Pearson Education, Inc., 2012.
12. L. Richardson, Sam Ruby, Mike Amundsen, RESTful Web APIs, O'Reilly, 2013.
13. F. Belqasmi, J. Singh, S.Y.B. Melhem, and R.H. Glitho, "SOAP-Based vs. RESTful Web Services: A Case Study for Multimedia Conferencing", IEEE Internet Computing, Vol. 16, No. 4, Pages 54-63, 2012.
14. V. Vernon, Domain-Driven Design Distilled, Addison-Wesley Professional, 2016.
15. E.J. Braude and M.E. Bernstein, Software Engineering: Modern Approaches (2nd Edition), Waveland Press, Inc., 2011.
16. I. Hughes and D. Immerman, "Ericsson shifts gears, introduces its IoT Accelerator and a new Experience Center", 451 Research, LLC, 2017.
17. M.E. Peck and D. Wagman, "Blockchains Will Allow Rooftop Solar Energy Trading for Fun and Profit", IEEE Spectrum, October 2017.