A Scale First approach for Web API development

Carlos Martín Flores González School of Computer Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica 2550–2254
Email: martin.flores@computer.org
Student ID: 2015183528

***Abstract*—Most applications using the Web as a delivery mechanism have had to embrace practices in order to constantly adapt their software so they can ensure their applications will be up and running most of the time. This is not an easy task and it becomes harder when applications are not designed to accommodate this change upfront. Web Application Programming Interfaces (APIs) have become in a popular solution to deliver data and content through the Web and, since they are available through Internet, they are exposed to the same concerns about how to manage change, particularly, the way the application could scale and manage new resources. Scale First is attempt to describe an approach where scalability topics are placed in the first place when implementing a Web API. It proposes a set of architectural, design and coding principles that allow to address change more effectively. These principles can be used in conjunction with software development methodologies to model the strategy of the system in terms of scalability and evolution.**

I. Introduction

In software development the usage of APIs is crucial to define protocols, operations and tools for building applications. Software developers access APIs as interfaces for code libraries to speed up development and/or take advantage of existing low level tasks. With the advent of the World Wide Web, the most successful distributed system, a new kind of API has emerged in order to provide services in applications through a set of Hypertext Transfer Protocol (HTTP) messages, the Web API. But, as with APIs, most of today's Web APIs have a big problem: once deployed, they can't change. In fact there are big-name APIs that stay static for years at a time, as the industry changes around them, because changing them would be too difficult [14]. The ability to adapt and evolve to new requirements it's crucial for a Web API to be successful under critical and high demanding scenarios, such as the ones that experience E-Commerce and data intensive applications.

> *"Most software systems are created with the implicit assumption that the entire system is under the control of one entity, or at least that all entities participating within a system are acting towards a common goal and not at cross-purposes. Such an assumption cannot be safely made when the system runs openly on the Internet."*
>
> — *Roy Fielding*
> *Architectural Styles and the Design of Network-based Software Architectures*

Although we know software can change, many Web APIs can't accommodate change easily because they were not designed and developed taking scalability in mind from the ground up. A scalable Web API is not only an application that has the ability to allocate new resources in order to support new demanding growth, but also able to decrease their resource consumption when demand is down. Scalability is one of the main concerns that organizations have to take in mind when implementing a software project, without it the ability to compete and bring services to a broader audience and even reputation is compromised.

Software methodologies don't propose a specific set of actions to take into consideration in order to address scalability concerns. Some agile methodologies such as Scrum and Extreme Programming (XP) encourages software projects to hit production or production-like environments as soon as possible(one of these approaches is the "Walking Skeleton", described by Allistair Cockburn [5] were a tiny implementation of the system is suppoused to perform a small end-to-end function), the main goal on those is to get rapid feedback about the deployment process and minimize further risks. In this particular we can say these methodologies are more intended to model the domain of the application rather than to model application growth.

In the other hand, modern engineering practices such as Continuous Integration and Continuous Delivery gives us a reference about how applications, their changes in the codebase and their deployment should be managed. In both practices, automation emerges as one of the key aspects since they rely on software to run test $\rightarrow$ build $\rightarrow$ acceptance $\rightarrow$ deploy cycles[1], this could allow not only faster response times during deployment but also better sense of quality [11]. In Continuous Delivery, deployment processes of applications might consider scalability concerns because the goal is to model a workflow where applications are always in a delivery-ready state and, in order to achieve it, software developers have to find the way to provide mechanisms where applications are available to be shipped to one or *n* servers at any time(ideally). An important thing to note here is that these practices are agnostic to the application they serve. This makes sense at first glance but at the same time it could lead to scenarios where monolithic[2] applications are trying to be deployed. Despite the success of monolithic applications, frustration increases especially when they are being deployed. Change cycles are tightly coupled, small changes requires the entire monolith to be rebuilt and

[1] In Continuous Delivery this workflow is known as a deployment pipeline.

[2] The term monolith has been in use by the Unix community for some time. It appears in The *Art of Unix Programming* to describe systems that get too big.

deployed. Over time it is often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource [10]. An application could be delivered fast but its internals doesn't allow a better way to scale it unless you try to scale the whole application at the same time. This approach could be expensive in terms of both money and resource consumption.

From the deployment standpoint, according to recent trends, organizations are planning for hybrid cloud computing solutions, meaning that increasingly they are relying their application deployment to more than one cloud computing provider [15] making even more important to have applications with cloud-friendly architectures. Here again, the eventually adoption of a development strategy which is aware about these kind of deployment considerations could make application distribution easily and prevent further headaches.

II. Web APIs

Web APIs is a term that refers to functions/operations that can be programmatically invoked using Web protocols. The term is based on the concept of Service Oriented Architecture (SOA) which proposed an attempt to change how software functionality is exposed through well-defined interfaces that were formally registered and could be discovered on the fly.

As the SOA notions matured, especially in their web services style and XML format of data, SOA evolved into more Web-friendly technologies such as Representational State Transfer (REST) and JSON data format that greatly simplified the reusability of an enterprise's capabilities. We use a common term, Web APIs, to refer to them.

The reason Web APIs are important even more today is that they allow organizations to make their business functionalities easily accessible to other entities (patterns, customers) and thus easily create new value-added services. Further, users have access to a computing device in the context of their daily lives like never before, i.e., mobiles. [19]

III. Scale First

Due the lack of a software methodology/approach able to embrace scalability concerns as a key role of the software development process of Web APIs from their initial designs, is that we introduce an approach where scalability becomes a feature of the application itself. By sticking to this approach, Web APIs will be able to evolve easier because growth aspects are attacked in the initial stages of the development process, making organizations and individuals even more aware about the entire ecosystem of the Web API.

In order to have *scale-ready* applications, Scale First proposes a set of principles for clean, well-structured and testable applications that will be able to evolve according their needs before a single line of code is written, here is where this approach differs from traditional methodologies because concepts such as software design and deployment are tightly coupled.

A. Principles in Scale First

- Resizable deployment environments
- Technology agnostic design
- Modular, scalable application design
  - Decompose application domain
  - Split conceptual model into separate models for update and display
- Testability and maintainability
- Automated deployment process
- Define a scalability policy

B. Resizable deployment environments

C. Technology agnostic design

The aim of technolody agnostic design(TAD) is to separate design and architecture from the technology employed and the specific implementation. This separation decreases both cost and risk while increasing scalability and availability of the Web API. With TAD both organizations and individuals are forced to create disciplines around scales that are not dependent upon any single provider or service. This discipline allows you to scale in multiple dimensions through multiple potential partners, the result of which is a more predictable scalable system independent of any single solution provider. A common misperception is that by implementing a certain solution –for instance a databse technology–, you are reliant upon that solution. Just because the project makes use of a trendy new database technology does not mean that the Web API is dependent upon it alone for scale. TAD impacts availability also. The most obvious is the way in which it supports the ability to switch providers of technology when one prover has significantly greater availability of quality than other providers.

1) TAD considerations for Web APIs: [1] proposes three steps for TAD designs. These steps are suitable for Web API projects as well and they can act as an initial help guide.

1) In the design itself, think in terms of boxes and wire diagrams rather than prose. Leave the detail of the boxes to the next step.
2) In defining boxes and flows, use generic terms. Aplication server, Web server, RDBMS[3], and so on. Don't use vendor names.
3) Describe requirements in industry standards. Stay away from proprietary terms specific to a vendor.

If the design is being pulled toward a vendor in a description or design statement, an attempt to "loose" that statement needs to be done to make it agnostic.

D. Modular, scalable application design

Much has been written and implemented in regards the benefits of modular and/or component-based architectures in software development. These principles and practices applies for the development of Web APIs too, but beyond this, the goal is to come up with an architecture that allows the Web API to become an autonomous component who communicate

[3]RDBMS stands for *Relational Database Management Systems*

with a mechanism such as web service requests, or remote procedure calls. That is make the Web API a service[4]. From the scalability standpoint, the main reasons for using services as components are the following:

- Technology Heterogeneity: Web APIs can be developed using the best suitable tool or technology stack depending of their needs.
- Only those services that need scaling will scale.
- Services are independently deployable: a change that is made to a single service is deployed independently without impacting any other system. This allows to get the code deployed faster. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve.
- Resilience: working as an isolated service if a Web API fails then that failure will not be propagated to other systems.

Notice the above are not the only characteristics of a service-oriented architecture, more of this is described in [13] and [10].

1) Decompose application domain: Domain context of the Web API has to be clearly delimited. The aim here is not to motivate the creation of very specific applications doing only just one thing, that could lead to management issues and since we are addressing the Web API development from the scalability standpoint, we are more interested in the creation of applications which represents a functional unit, that way logic and features inside can be managed and shared easily.

The work on [8] provides a guide on how to create systems that model real-world domains. It introduces the concept of *bounded context*: "a specific responsibility enforced by explicit boundaries". The idea is that any given domain consists of multiple bounded contexts, and residing within each are things that do not need to be communicated outside as well as things that are shared externally with other bounded contexts. Each bounded context has an explicit interface, where it decides what models to share with other contexts. To get information from a bounded context, or want to make requests of functionality within a bounded context, a communication with its explicit boundary using models is performed.

When developing a Web API, the domain could be separated in bounded contexts meaning that each context has the potential to become separate Web API. This separation process is crucial, because no matter if in the first stages of the Web API development bounded contexts are not meant to be separate APIs, but depending of application needs they can be extracted and implemented as a separate API. With bounded contexts in place, future scalability efforts could be performed faster so this is a very important activity to run when designing and develop applications that are meant to be subject of change. In the fictional application design on figure 1, although a Web API could be developed using the entire domain as a reference, both Sales Context and Support Context are potential separate Web APIs.

[4]Or a *microservice*, an architectural style to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [10].

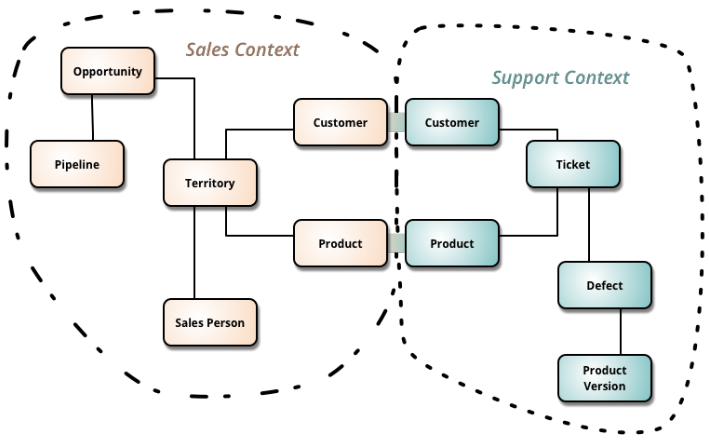

Figure 1: A fictional application domain with two bounded contexts: Sales Context and Support Context. [9]

2) Split conceptual model into separate models for update and display: Depending of the domain, some applications can be more intensive in reading(query) than in writing data and viceversa even so both responsabilities have to be identified and separated into models for update and display, ideally each model will hold a set of autonomous components(such as databases) so each model is not depending of the other. This separation allows applications to scale depending of the demand of each of its responsabilities, that way if in a Web API the most intensive part is related with the query of data sources for display purposes then this portion of the Web API can be isolated and segregated in another Web API whose only purpose will be to serve queries. Since queries are now being performed off of a separate data store than a master database, and assuming that the data that's being served is not 100% up to date, it can easily to add more instances of these stores without worrying that they don't contain the exact same data. This gives cheap horizontal scaling for queries. Also, since your not doing nearly as much data transformation, the latency per query goes down as well. Simple code is fast code [6].

An often implemented solution to this is the three-tiered application in which there is a separate data, logic and presentation tier. Within this solution, the database in the data tier is often seen as one CRUD (Create, Read, Update and Delete data) data store in which all commands and queries are performed on the same database. This can lead to locking, performance and scalability problems, especially with larger commands or queries, since all things have to be taken care of sequentially. Distributing parts of the system in combination with selective locking of data provides a partial solution, but leads to a high probability of data inconsistency. An option is to split parts of the system that have an emphasis on consistency from parts that should have an emphasis on availability or partition tolerance. The Command Query Responsibility Separation (CQRS) is a pattern in which all logic of a software product is separated based on whether it changes the application state (commands) or only queries it (queries). This means executing commands is done by different components than the one responsibly for executing the querying tasks, all of which can be done distributed and in parallel. Besides helping to solve the scalability problem of multi-tiered software products by enabling architects to distribute tasks of the system among an unlimited amount number of systems, CQRS also helps to implement a higher level of variability in online software products. The high level of variability is caused by the fact the main pattern keeps commands strictly separated from queries and has a large collection of sub patterns using the distributed nature of the pattern to enable, among others, all sort of different tenant dependant configurations, work flows, and business rules [12].

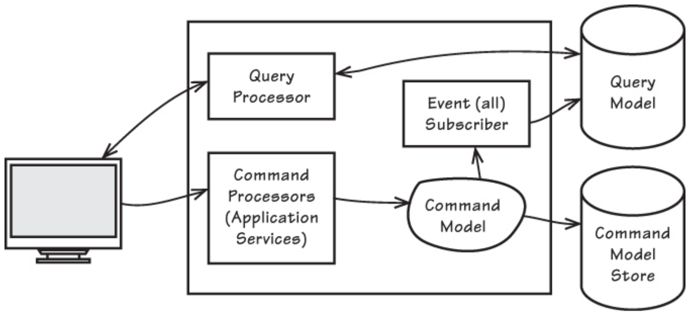

Figure 2: With CQRS, commands from clients travel one way to the command model. Queries are run against a separate data source optimized for presentation and delivered as user interface or reports [20].

With a CQRS approach like the one in figure 2 could led a Web API to be separated in two subapplications, each application can growth independently so resources can be assigned depending of its demand.

E. Testability and maintainability

A test strategy for a Web API that is intented to evolve quickly must provide the following:

1) Test must run fast
2) Test must cover as much of application
3) Test must be automated
4) Must provide support to continuous integration

In the above list, items 1, 2 and 3 are intented to describe how test should behave while item 4 is intented put a context about how tests should be executed. Although many kinds of testing exists [11], testing for scaling fast and reliable must have the automated component. For Web APIs, unit, integration and acceptance tests are the preferable set of tests that should be included.

The idea behind item 4, continuous integration is that, if regular integration of your codebase is good, why not do it all the time? In the context of integration, “all the time” means every single time somebody adds any change into the system [3]. The application of this concept in the Web API by the usage of continuous integration software[5] in the development process, will allow to deliver software much faster, and with fewer bugs[6].

[5]Jenkins, Bamboo, Cruise Control, SnapCI, TravisCI

[6]Or with no bugs at all

To increase the Web API autonomy and increase overall availability, is also needed to identify and repair problems, and then notify the appropriate system operator about the service’s current status. Web API has to provide support for actively monitors its internal state, acts on potential trouble, tries to heal itself, and continuously publishes its status is required also. For this, [16] proposes the implementation of a service watchdog[7], a pattern where a service actively monitors its internal state, acts on potential trouble, tries to heal itself, and continously published its status. Web APIs on *Scale First* must contain a component in charge of monitoring the API’s state. This component publishes the API’s state periodically, and also when something meaninful occurs.

F. Automated deployment process

Scale First relies in the concepts and principles behind Continuous Delivery [11] in order to define processes that will allow to have a Web API application in a delivery-ready state. To achieve this, continuous delivery introduces the concept of *deployment pipeline* which is an automated manifestation of the process for getting software from version control into the users. Every change of the software goes through a complex process on its way to being released. That process involves building of the software, followed by the progress of these builds through multiple stages of testing and deployment. The deployment pipeline models this process, and its incarnation in a continuous integration and release management tool is what allows to see and control the progress of each change as it moves from version control through various set of tests and deployment to release to users. A figure representing a basic pipeline can be found in appendix A.

An automated deployment process for a Web API in *Scale First* must include tasks in which new resources are able to be both, allocated and deallocated according application demands. This ability of making applications and environments adjust to new conditions is what we refer as the *scale-ready* attribute in this Web API development approach.

G. Define a scalability policy

IV. Experimental Setup

The exploratory study of the impact of the *Scale First* approach for Web API development could be composed of four parts. In the first part we can collect a set of Web API projects in which one or more of the principles of the Scale First approach are not adopted. In the second part, the development of a non-trivial Web API that follows the Scale First approach is required, this allow us to identify the potential of Scale First. In the third part, we need to create and exercise scenarios where scalability is required on both types of applications: the ones that are not following Scale First and the one who does it. Lastly we measure and interpret the impact of Scale

[7]*Watchdog* is a term borrowed from the embedded systems world. A watchdog is a hardware device that counts down to 0, at which point it takes action, such as resetting the device. To prevent this reset, the application has to “kick the dog” before the timer runs out. If the application doesn’t reset the counter, it could mean that the application has stopped responding. A reset would fix that.

First by analyzing the response time involved to perform the scalability task and the assigned resources to it such as money, hardware and people.

APPENDIX A
A BASIC DEPLOYMENT PIPELINE

REFERENCES

[1] M. Abbot and M. Fisher, *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise*, 1st ed. Massachusetts, United States: Person Education, 2010.
[2] M. Abbot and M. Fisher, *Scalability Rules: 50 Principles for Scaling Web Sites*, 1st ed. Massachusetts, United States: Person Education, 2011.
[3] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change.* 2nd edition, Addison-Wesley, 2004.
[4] L. Chen, *Continuous Delivery: Huge Benefits, but Challenges Too*, IEEE Software, vol. 32, no 2, 2015, pp. 50-54.
[5] A. Cockburn, *Walking skeleton*. Available at: http://alistair.cockburn.us/Walking+skeleton
[6] U. Dahan. *Clarified CQRS*. December, 2009. Available at: http://udidahan.com/2009/12/09/clarified-cqrs/
[7] E. Espinha, A Zaidman and HG Gross *Web API Growing Pains: Stories from Client Developers and Their Code*, Delf University of Technology, The Netherlands. 2014
[8] E. Evans. *Domain-Driven Design*. Addison-Wesley. 2003.
[9] M. Fowler. *BoundedContext*. January, 2014. Available at: http://martinfowler.com/bliki/BoundedContext.html
[10] M. Fowler, *Microservices*. Available at: http://martinfowler.com/articles/microservices.html
[11] J. Humble and D. Farley, *Continuous delivery : reliable software releases through build, test, and deployment automation*, 1st ed. Massachusetts, United States: Person Education, 2011.
[12] J. Kabbedijk and S. Jansen and S. Brinkkemper. *A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software*. Proceedings of the 17th European Conference on Pattern Languages of Programs. 2012.
[13] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media Inc, 2015
[14] L. Richardson and M. Amundsen, *RESTful Web APIs*, 1st ed. California, United States: O'Reilly Media Inc, 2013.
[15] RightScale. *RightScale 2015 State of the Cloud Report*. RightScale, Inc. 2005. Available at: http://www.rightscale.com/lp/2015-state-of-the-cloud-report?campaign=701700000012UP6
[16] A. Rotem-Gal-Oz. *SOA Patterns*. 1st. Manning Publications. 2012.
[17] N. Serrano, G. Gallardo and J. Hernantes, *Infrastructure as a Service and Cloud Technologies*, IEEE Software, vol. 32, no 2, 2015, pp. 30-36.
[18] T. Schlossnagle, *Scalable Internet Architectures*, O'Reilly Open Source Convention 2010. O'Reilly Media Inc, 2010.
[19] B. Srivastrava. *Componsing Web APIs: State of the Art and Mobile Implications (Tutorial)*. Proceeding MOBILESoft 2014 Proceedings of the 1st International Conference on Mobile Software Engineering and Systems. 2014.
[20] V. Vernon. *Implementing Domain Driven Design*. 1st ed. Addison-Wesley. July, 2013.

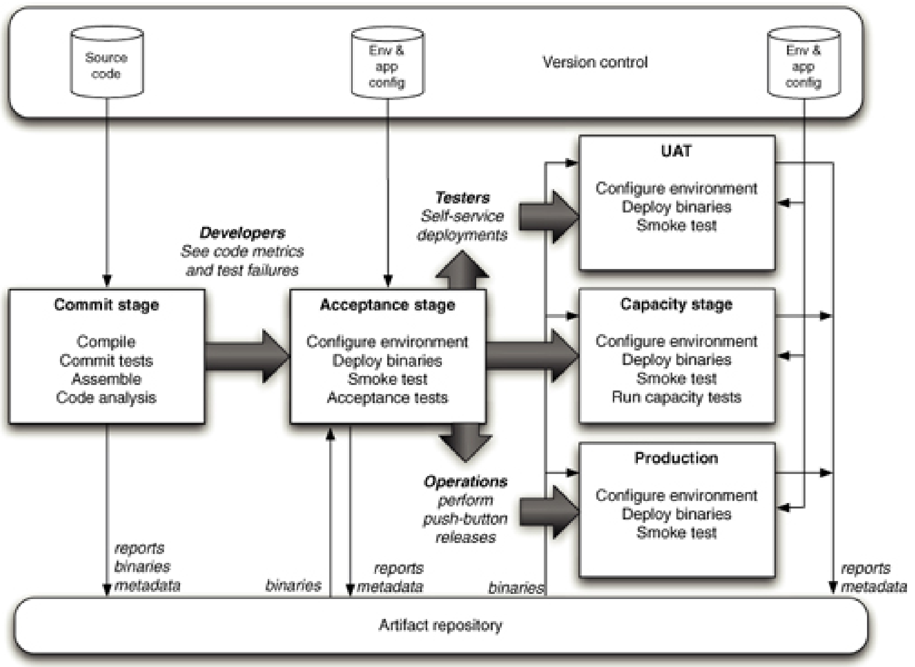

Figure 3: A basic deployment pipeline