Designing To Be Replaced: Highly Configurable API Gateway Framework to Facilitate Lightweight Microservices

Naren Sivagnanadasan
ACM@UIUC - University of Illinois Urbana-Champaign

bcongdo2
ACM@UIUC - University of Illinois Urbana-Champaign

ssapra2
ACM@UIUC - University of Illinois Urbana-Champaign

Aashish Kapur
ACM@UIUC - University of Illinois Urbana-Champaign

Taeyoung Kim
ACM@UIUC - University of Illinois Urbana-Champaign

ABSTRACT

We introduce a novel framework for creating unified application/programmer interface (API) layers for the microservice architecture. We show that because of this new framework called Arbor, we are able to allow a whole new class of developers to create their infrastructure in the microservice paradigm and in particular allow for high churn development groups (research groups and university institutions with dependence on student developers for instance) to continually update their software infrastructure, without significant additional documentation and refactoring.

1. INTRODUCTION

1.1 Motivation

The impetus to create such a framework stems from the reality that the majority of developers who work in our development group, will not be there in 4 years (and most will leave earlier). We also see new developers walking in oblivious of the frameworks we have used in the past. Just in the past four years, Rails has been supplanted by Node.js and Angular/React, and PHP has gone from a "goto" language to an outdated language with only a few purposes.

This poses a few issues to these high churn development groups when considering choices for technology stacks in the context of maintainability and future-proofing. Pragmatically, original authors of a project for a student club, for example, will not be in school for more than a few years to maintain the project, and the newcomers will not know how to sustain the technology stack of the project created just a couple years ago.

While monolithic architectures have their benefits such as clear model ownership (a-write-once-use-everywhere mentality), the resultant complexity of highly intertwined dependencies coupled with the high churn rate in maintainers make it challenging to maintain such a project. Development efforts therefore are highly duplicate as it was many times easier to develop independent full stack systems for different tasks then trying to integrate new services into a unified infrastructure. This also inhibits the adoption of new platforms as a developer who is trying to create a new client (e.g. for mobile) in a monolithic environment may have to contend with unification of large independent services on the client side and the difficulty of shimming in a new client into the tightly integrated infrastructure .

SUSTAINABLE INFRASTRUCTURE

Microservices

The response from many figures in the software industry (e.g. Netflix and Amazon) to this problem of an unmaintainable infrastructure was to adopt the aforementioned microservice architecture, which looks to promote modularity and minimization of external dependencies.

The system breaks down into two major groups: clients and services. All components of the infrastructure are completely separate applications running on separate processes in deployment.

This approach provides several key benefits:

1. Services are fair simpler than their monolithic counterparts. Applications that used to manage numerous user models, the intertwining middleware and the API now typically manage one or two models with simple middleware to expose the data in an API.
2. Data dependencies can now be resolved through API calls to other services.
3. Any service can be discarded and replaced without affecting other services, providing functionality parity and modularity.

4. Clients are far simpler because the much of the application logic has been abstracted behind the gateway.

5. Service maintainers now control small and manageable isolated components and therefore can focus and own their own codebase.

API Gateway

Many implementations of the microservice architectures include what is referred to as an API Gateway.

<INSERT MORE ON THE CONCEPT OF AN API GATEWAY>.

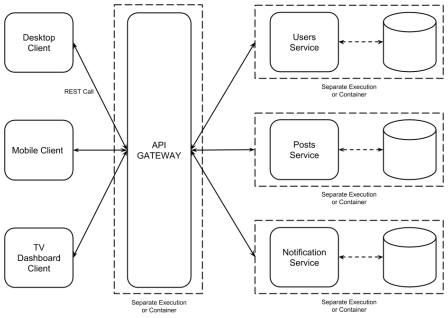

Figure 1. Example of an Infrastructure implemented with microservices using an API gateway

There are many platforms which provide the functionality of an API gateway; Amazon Web Services, Nginx, and others have offerings that manage access to services and coordinate microservice systems. There also exists a few frameworks to self-host an API Gateway. We will examine Tyk and Kong, two popular options, however, there are others that are language, platform or web framework specific.

Tyk

Tyk is a full-featured gateway that looks centralized much of the common tasks that exist within common infrastructure setups. It does things like having a centralized Authentication provided in the gateway, allows for granular access control to routes and hosts API documentation. It configures its system using a series of JSON objects either held in a database or in a series of files that are parsed and include many options controlling the massive feature set Tyk offers.

Kong

Kong also has a vast set of features and plugins, as also looks to centralize much of the common tasks in managing the infrastructure

Kong, however, takes a different approach to its proxy. It only takes a hostname in and relies on the client to provide the host it's looking for (it does allow for a host proxy though the documentation is not clear as to how that works exactly). Since everything is done through DNS listings, it is very easy to add your service, but it means that it's hard to come in and see what is going on without the use of its admin API

There are many other platforms that include gateway functionality, in addition to other features, such as Docker and Kubernetes. Compose. However, some of these systems are high performance, complex pieces of software that are meant to serve millions of users at once, which is a bit heavy handed for nonindustrial use. Additionally, the use of these systems may require platform buy-in, suffer from lack of ease-of-use and clarity and perhaps require the user to spend money to set up a gateway.

Security and Vunerability

To prevent client and service developers to manage an ever-increasing list of services and their APIs, the API gateway exposes all the services as a unified RESTful API while managing application level security, inter-service communication, and routing of requests.

Third party APIs are easily and securely accessible by any client that can access the API Gateway. This moves the burden off from the client developer to juggle tokens.

From a different perspective, a security exploit on the API Gatway can affect multiple microservices

Scalability

ARBOR

We were not happy with the current offerings for API Gateways as many were just too complex to maintain in our organization with churn being a big factor. We, therefore, created our own API Gateway framework which we call Arbor.

Arbor is a statically configured API Gateway framework written in Go. It provides the key gateway services, having a full-featured REST proxy all behind its hostname, with websockets and more modern protocols in the pipeline, it manages application level security and simple request sanitization, it also features a human-readable service definition format.

Arbor is broken into four major components: the server, proxy, services registry, and security. The full system architecture is described in Figure 2. and is centralized around the Arbor server. The server manages the execution of the subsystems within Arbor.

The first layer on the server is the router. As of now, the router is vanilla gorilla/mux but we fully expect to need to modify the router to support future development.

Route definitions are user-defined through two types defined by Arbor/services, Routes and RouteCollections, in which Routes define a specific route, and RouteCollections encompass a service's routes. As part of the service definition, the

user specifies the proxy handler that would manage the communication between the client and the service from the set of functions in the proxy package.

At runtime these service definitions are loaded into the router and as a request comes in the router hands the request to the appropriate proxy handler.

Before any further processing, if the user enables security features, the next step for the request is to go through two main security features. First is a client authorization verification step. When security features are enabled the gateway requires clients to provide a token generated by the security package. If this is not provided the request is denied at the gateway level. Any successful access is logged in a access log file and then the request is moved on to the second step. We wanted to take the burden off our service and client developers to do proper html sanitization, so we have a module in the pipeline to clean the request. Services are still required to do some validation but we are looking to centralize common tasks.

Afterwards, the request is sent to the proxy server, which will inject any required access keys or extra data in order to fulfill a request. This allows clients to avoid needing to manage a huge set of tokens and keys. The request goes out from the proxy server and then the response is forwarded back to the client.

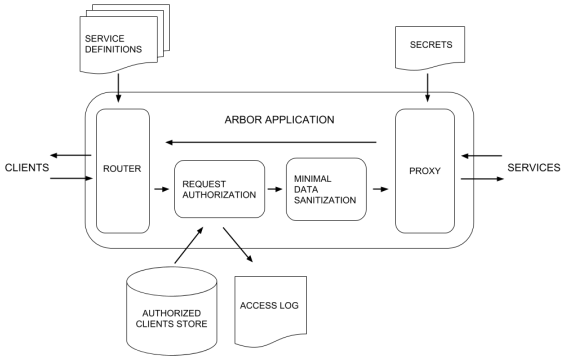

Figure 2. Arbor Architecture

Creating an API Gateway with Arbor

Below we outline a full working API Gateway for managaging two services using Arbor

This means that registering a new service is just adding a file, recompiling and deploying, and authorizing a client is a just simple command as of now. The question of what data is available is answered by the definition itself. An full API gateway example is detailed below. There are a

Arbor attempts to be simple, making very few assumptions, and is easily extensible.

```
//gateway/users.go
package gateway

import (
    "net/http"

    "github.com/acm-uiuc/arbor"
    "github.com/acm-uiuc/groot-api-gateway/config"
)

//Location
const UsersURL string = config.UsersURL

//Service Data Type
const UsersFormat string = "JSON"

//API Interface
var UsersRoutes = arbor.RouteCollection{
    arbor.Route{
        "NewUser",
        "POST",
        "/users",
        NewUser,
    },
    arbor.Route{
        "UsersLogin",
        "POST",
        "/users/login",
        UsersLogin,
    },
    arbor.Route{
        "UsersLogout",
        "POST",
        "/users/logout",
        UsersLogout,
    },
}

// arbor.Route handler
func NewUser(w http.ResponseWriter, r *http.Request) {
    arbor.POST(w, UsersURL+r.URL.String
(), UsersFormat, "", r)
}

func UsersLogin(w http.ResponseWriter, r *
http.Request) {
    arbor.POST(w, UsersURL+r.URL.String
(), UsersFormat, "", r)
}

func UsersLogout(w http.ResponseWriter, r *
http.Request) {
    arbor.POST(w, UsersURL+r.URL.String
(), UsersFormat, "", r)
}
```

Figure 3. This is a caption

```
//gateway/products.go
package gateway

import (
    "fmt"
    "net/http"

    "github.com/acm-uiuc/arbor"
    "github.com/acm-uiuc/arbor/proxy"
    "github.com/acm-uiuc/arbor/security"
)

//URL of the Product Service API
const productServiceURL string = "http://0.0.0.0:5000"

//Data format of the API
const productServiceFormat string = "JSON"

//Collection of routes to expose
var productServiceRoutes = arbor.RouteCollection{
    //Route definition
    arbor.Route{
        Name:    "CreateProduct",
        Method:  "POST",
        Pattern: "/product",
        Handler: createProduct,
    },
    arbor.Route{
        Name:    "GetProduct",
        Method:  "GET",
        Pattern: "/products/{id:[0-9]+}",
        Handler: getProduct,
    },
}

//Handlers
func createProduct(w http.ResponseWriter, r *http.Request) {
    //proxy (ResponseWriter, URI, format, ApplicationAccessToken, Request)
    arbor.POST(w, productServiceURL+r.URL.String(), productServiceFormat, "", r)
}

func getProduct(w http.ResponseWriter, r *http.Request) {
    //proxy (ResponseWriter, URI, format, ApplicationAccessToken, Request)
    arbor.GET(w, productServiceURL+r.URL.String(), productServiceFormat, "", r)
}
```

Figure 4. This is a caption

RESULTS AND EVALUATION

This new framework and architecture choice has accelerated the creation of new pieces infrastructure in our organization many times over. It has given us the freedom to replace many aging systems with new modern ones without fear of breaking other things. It also gives the developers the leeway to focus more on functionality over sustainability, since it is now ex-

```go
//main.go
package main

import (
    "fmt"
    "net/http"
    "./gateway"
    "github.com/acm-uiuc/arbor"
    "github.com/acm-uiuc/arbor/security"
)


func RegisterServices() arbor.RouteCollection {
    Routes = append(gateway.UserRoutes, gateway.ProductRoutes...)
    return Routes
}

//Arbor configurations
func ConfigureArbor() {
    //Location for the Access Log
    security.AccessLogLocation = "/tmp/arbor_access.log"
    //Location for the Client Registry
    security.ClientRegistryLocation = "/tmp/arbor_clients.db"
}


func main() {
    //Configure Arbor
    gateway.ConfigureArbor()
    //Register the Routes in a Collection and Boot Arbor
    arbor.Boot(gateway.RegisterServices(), "0.0.0.0", 8000)
}
```

Figure 5. This is a caption

pected their code will last perhaps two or three years max. Each service can now range from 100 lines of code to 1000. Our infrastructure is now language agnostic as well. We have a ruby, a python and a javascript developer all developing code for the same system. It is also easier now to replace a system since a spec of what needs to be supported is kept in the schema file for that service. We now have 15 services and two clients all managed through Arbor serving 1000 people.

Future Work

Throughout Arbor's development, we have uncovered a few issues which should be identified and addressed:

1. Arbor currently only supports RESTful APIs, we are looking to add support for WebSockets so that the final parts of our infrastructure can come under the gateway umbrella.

2. Documentation generation from parsing schema files would be a welcomed feature so unaffiliated consumers of our API will have documentation other than our repo

3. Improved centralized functionality for things like security is being considered. Currently, there is a simple sanitizer in the proxy pipeline that can be enabled. We have this so that service developers do not need to consider security other than that which is specific to their service. This sanitizer is not very powerful at this point and we must also balance the urge to centralize functionality with not making assumptions about development processes.

CONCLUSION

In this paper, we have presented our new framework for API Gateway definition. It is to be seen as a Sinatra (Flask) to Kong or AWS's Rails (Django). We do not expect Arbor to be serving millions, but we think that this framework will let a whole new set of developers experiment with microservices. It looks to provide the barebones API Gateway functionality and let the developer focus on the application functionality. Ultimately for us, it means we no longer have to race against time to train developers before the lead maintainers leave and that the infrastructure of our organization can now better reflect the needs of its members. We are now more open to one-off contributors, who wish to add one service and leave. When the next web framework of choice emerges, we will have a system ready to accept it with minimal wrestling. The source code is released under the University of Illinois/NCSA Open Source License and is available at **`https://github.com/acm-uiuc/arbor`**. The results of the architecture change can be seen at our new site at **`https://acm.illinois.edu`**

ACKNOWLEDGEMENTS

Project Groot is supported by Amazon.com Inc. Palantir Technologies and the Dept. of Computer Science at the University of Illinois Urbana-Champaign. ACM@UIUC is funded by industry sponsorships and grants from University of Illinois Urbana-Champaign, and we are grateful to our sponsors for helping us to solve challenging problems. To see the full list of ACM@UIUC's sponsors, please visit our website.