DZone

THE 2018 DZONE GUIDE TO

API Management

COMPARATIVE VIEWS OF REAL WORLD DESIGN

VOLUME V

BROUGHT TO YOU IN PARTNERSHIP WITH

ca technologies

Runscope

WSO2

Dear Reader,

Welcome to DZone's 2018 Guide to API Management. Put your ear to the ground; hear it. That's not just a train coming. It's a train that's moving full steam ahead, and its cargo is connected devices and services. From the ubiquitous smart phone to the sensors in your key fob, from beacons on the cargo train to automated banking. All of these are connected by an ever-growing number of Application Programming Interfaces. These APIs are the synapses that allow our interconnected world to communicate.

With that growth there comes the need to service, secure, and evolve in an always-on world. In this guide, we will focus on API management patterns, managing SDKs across multiple languages, APIs and their role in microservices, security, best practices, and a general overview of their evolution.

From the first parameter born from the need for an encapsulated piece of re-usable code, a function call was born. Since then, programmers have been iterating on this paradigm. From sockets and COM, to SOAP and REST, the remote procedure call continued to grow. This has evolved further to include bi-directional communication, the need to push messages to your nodes. Low power devices and IFTTT (If This, Then That) protocols have been layered on to streamline communication through the APIs. Staying on top of all this technology and growth is key to success.

Today's customers require APIs to enable all of these scenarios. They require these APIs to be always on. They require them to be updated. They expect them to be secure. And they expect them to be supported with proper SDKs and documentation. Failure in any of these areas can slow your adoption or worse, lead to a crippling security breach. DZone's API Management Guide will help you understand the best practices and prepare for this connected world.

Thank you for downloading this guide. As the proliferation of technology continues to expand and weave itself into the fabric of our lives, it will be critical for the industry to stay on top of the latest API and software technologies. Be sure to check back at DZone to catch up with latest in API management and enterprise integration. As with any great community, your feedback matters, and we would love to hear it. Thank you.

By Rick Severson
DIRECTOR OF ENGINEERING, DZONE

Table of Contents

DZone is...

BUSINESS & PRODUCT

MATT TORMOLLEN
CEO

MATT SCHMIDT
PRESIDENT

JESSE DAVIS
EVP, TECHNOLOGY

KELLET ATKINSON
MEDIA PRODUCT MANAGER

PRODUCTION

CHRIS SMITH
DIRECTOR OF PRODUCTION

ANDRE POWELL
SR. PRODUCTION COORD.

ASHLEY SLATE
DESIGN DIRECTOR

G. RYAN SPAIN
PRODUCTION COORD.

BILLY DAVIS
PRODUCTION COORD.

NAOMI KROMER
SR. CAMPAIGN SPECIALIST

JASON BUDDAY
CAMPAIGN SPECIALIST

MICHAELA LICARI
CAMPAIGN SPECIALIST

EDITORIAL

CAITLIN CANDELMO
DIR. OF CONTENT & COMMUNITY

MATT WERNER
PUBLICATIONS COORDINATOR

SARAH DAVIS
PUBLICATIONS ASSOCIATE

MICHAEL THARRINGTON
CONTENT & COMMUNITY MANAGER II

KARA PHELPS
CONTENT & COMMUNITY MANAGER

TOM SMITH
RESEARCH ANALYST

MIKE GATES
CONTENT TEAM LEAD

JORDAN BAKER
CONTENT COORDINATOR

ANNE MARIE GLEN
CONTENT COORDINATOR

ANDRE LEE-MOYE
CONTENT COORDINATOR

LAUREN FERRELL
CONTENT COORDINATOR

LINDSAY SMITH
CONTENT COORDINATOR

SALES

CHRIS BRUMFIELD
SALES MANAGER

FERAS ABDEL
SALES MANAGER

JIM DYER
SR. ACCOUNT EXECUTIVE

ANDREW BARKER
SR. ACCOUNT EXECUTIVE

BRETT SAYRE
ACCOUNT EXECUTIVE

ALEX CRAFTS
KEY ACCOUNT MANAGER

JIM HOWARD
KEY ACCOUNT MANAGER

BRIAN ANDERSON
KEY ACCOUNT MANAGER

SEAN BUSWELL
SALES DEVELOPMENT REPRESENTATIVE

MARKETING

SUSAN WALL
CMO

AARON TULL
DIRECTOR OF MARKETING

SARAH HUNTINGTON
DIRECTOR OF MARKETING

KRISTEN PAGÀN
MARKETING SPECIALIST

COLIN BISH
MARKETING SPECIALIST

Executive Summary

BY MATT WERNER - PUBLICATIONS COORDINATOR, DZONE

APIs have consistently driven the development of new solutions and growth of existing applications for decades. They're essential for developers to make their own software better and to make other software better by extension. That doesn't mean there isn't room to improve how APIs are secured or their ease of adoption. To learn more about out why developers use and create APIs, DZone surveyed 712 audience members about their opinions on APIs how they have integrated services.

AS EASY AS API

Data: Survey respondents reported that simplicity (70%), performance (65%), consistency (58%), and documentation (50%) are the most important attributes of a quality API.

Implications: Above all, developers are looking to get started with new APIs as quickly as they can. API providers should make sure that their APIs are easy-to-use and reliable, and have documentation that can be referred to in case users have a question.

Recommendations: Encouraging API use is a major part of building communities around enterprise software, and if developers cannot quickly make sense of it, they will quickly abandon the project unless it's a business-critical issue. To avoid losing potential new users that could innovate your product, encourage API creators to thoroughly document their work and compile it in an easy-to-search format so even if using the API is complex, there are extensive instructions for using it.

DEVELOPERS AT REST

Data: When building APIs, 52% of respondents use REST whenever possible and 33% use REST except for specific instances. Only 3% of respondents deliberately avoid using REST.

Implications: Representational State Transfer (REST) is one of the most popular ways to build web services today, and the vast majority of developers default to REST when they build new APIs.

Recommendations: REST systems are best known for high performance and the ability to re-use components without affecting an entire application or system. It's currently considered a standard for application development, and only in legacy applications that are using older standards such as RPC is REST deliberately avoided. That being said, it is important to understand the history of integration, as detailed in Fernando Doglio's article "Evolution of Systems Integration Before REST" in this Guide.

THE WHY OF APIS

Data: 62% of survey respondents implement APIs to allow customers to integrate their software with the API provider's, 42% want to encourage their users to find new or innovative solutions using the software, 37% implement them for mobile device support, 22% seek to monetize their APIs, and 21% implement them for IoT device support.

Implications: Unsurprisingly, most companies implement APIs to allow customers to make their lives easier by developing their own integration solutions. However, there is also a significant portion that use APIs to support different mobile or IoT devices without working on porting the application for each kind of device platform.

Recommendations: APIs can make everyone's lives easier, from customers to developers. They can be key to growing the market share of your applications, but they also must be secured, as many data breaches such as those suffered by Delta Airlines, Panera Bread, and Sears happened because of insecure APIs. Guy Levin's article "The Role of API Gateways in API Security" can offer some insight into making sure your APIs benefit your organization without falling victim to common attacks.

Key Research Findings

BY JORDAN BAKER - CONTENT COORDINATOR, DZONE

DEMOGRAPHICS

For this year's API Management Guide survey, we received 1,287 responses, with a 55% completion rating. Let's quickly break down the demographics of these survey respondents.

- The average respondent has 14 years of experience in the software development field.
- 34% work as developers/engineers, 22% as developer team leads, and 21% as architects.
- The most popular programming language ecosystems are:
  - 81% - Java
  - 68% - JavaScript (client-side)
  - 41% - Node.js
  - 36% - Python
- 64% use Java as their primary language at work, followed distantly by C# at 8%, and JavaScript/Node.js at 7%.
- 80% are currently developing web applications/services (SaaS) and 50% are currently developing enterprise business applications.
- 22% work for companies sized 1,000-9,999 employees, 20% for companies sized 10,000+, and 16% for companies with 100-499 employees.

THE NEED FOR GREATER SIMPLICITY

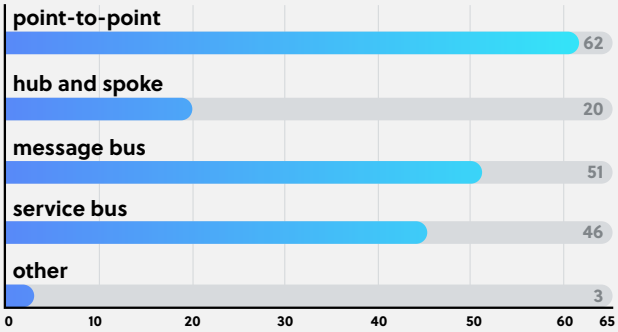
73% of survey respondents spend over a quarter of their time working on integration in some capacity during a given project. 43% reported spending somewhere between 25-50% of their time working on integration, while another 11% told us that they spend between 50-75% of development time on integration. Those are staggering numbers. Interestingly, when we look at the time spent on integration as reported by both web application/services developers and enterprise business application developers, the numbers are extremely similar. 43% of respondents who work as web application/service developers reported spending 25-50% of their time on integration, 38% spend less than 25%, and 11% spend 50-75% of their time on integration. Among respondents who work as enterprise business application developers, 48% spend somewhere between 25-50% of their time on integration, 31% spend less than 25% of their time on integration, and for 11% integration takes up 50-75% of their time.

To delve even further into the issue of time consumption, let's look at the most time-consuming levels of the integration process. When we asked respondents to describe the most time-consuming levels of the integration process, 62% said the application/communication level, 44% reported the method/business processes (e.g. allowing multiple applications to access one another's methods), 43% said the data level, and 37% reported UI (e.g. presenting multiple applications to users at once). Furthermore, these frustrations seem ubiquitous across programming language ecosystems. When we compare these time consumption statistics to respondents working in the Java and client-side JavaScript ecosystems (the two most popular ecosystems in this survey), we see the following breakdown:

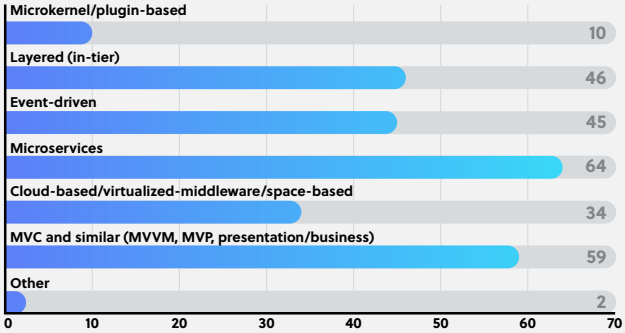
- JavaScript
  - 63% - Application/communication level
  - 46% - Method/business process
  - 44% - Data level
  - 40% - UI
- Java
  - 65% - Application/communication level
  - 44% - Method/business process
  - 41% - Data level
  - 36% - UI

Thus, we can surmise that similar time-consumption patterns apply to developers across languages and ecosystems.

WHAT INTEGRATION TOPOLOGIES DO YOU USE?



WHAT SOFTWARE ARCHITECTURE PATTERNS AND STYLES DO YOU USE?



Given the large amount of time and effort put into the integration process, it should come as no great surprise that respondents told us that simplicity is the biggest factor when working on integration projects and developing with APIs. Let's again turn to our web application/service and enterprise business application developers. Among web app/service developers, 59% reported simplicity as the biggest factor they look for when approaching integration problems. Additionally, 72% of web app/service developers chose simplicity as the most important API quality attribute to consider when using an API, and 64% chose simplicity as the most important API quality attribute when building APIs. These patterns hold true for the enterprise business application developers who took this survey. Among this group, 73% noted simplicity as a large factor in approaching integration issues, 67% chose performance/robustness, and 60% said consistency.

INTEGRATION, APIS, AND OPEN-SOURCE TOOLS

Now that we have an understanding of some of the hurdles that developers face with APIs and why API integration simplicity is so important, let's take a look at how developers are using APIs. When it comes to the types of systems that get integrated, 37% of respondents integrate mobile systems, 37% integrate BI/analytics systems, and 35% integrate CRM systems. As these were the three most popular responses among this year's survey respondents, let's use these systems as a means for comparison.

When we look at why developers and/or organizations implement APIs, allowing customers to integrate with their software came in as the top concern for developers and organizations integrating for mobile (57%), BI/analytics (59%), and CRMs (55%). This is where the unanimity ends, however. The second highest concern among respondents working with mobile integration was the need for mobile support (49%), whereas BI/analytics (42%) and CRM (41%) developers told us encouraging users to develop new solutions using their software was the second largest reason for implementing APIs.

Comparing these same three systems to the approaches to application integration survey respondents use, we again see some interesting trends. The top approach to application integration proved the same among these three categories. 77% of respondents working with mobile system integration, 75% of respondents working with CRM integration, and 74% of respondents working with BI/analytics integration told us that sending messages among applications was their top approach to application integration. And, again, respondents' choices diverged after the top choice. 61% of mobile respondents reported sharing a common database among different applications as their approach to application integration, whereas 59% of BI/analytics developers and 59% of CRM developers reported that transferring files among different applications was their preferred approach.

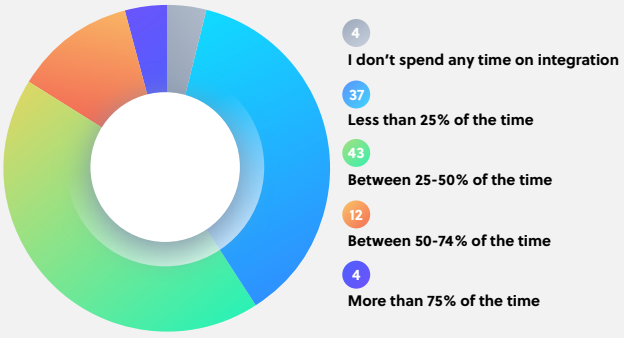
Now that we've covered the systems and APIs respondents use, let's take a quick look at the popularity of open-source frameworks and tools. While the popularity of open-source tools is not in itself surprising, it is intriguing that for every tool we asked survey takers about (save iPaaS solutions), open-source solutions took the cake. The two most popular integration frameworks, suites, or ESBs reported by respondents were Spring Integration (40%) and Apache Camel (24%). For messaging tools, the top three choices reported by respondents were, again, all open-source. Apache Kafka (39%) proved the most popular, followed closely by RabbitMQ (36%) and ActiveMQ (28%). And Swagger was the most popular framework among respondents for designing and documenting RESTful APIs (45%) as well as API management (68%).

RESTFUL APIS AND DATA EXCHANGE FORMATS

Representational State Transfer, or REST, has become an extremely popular architecture with which to develop APIs. Among survey respondents, 52% use REST as much as possible, and 33% use REST whenever possible, with a few exceptions, when building and designing APIs. Only 3% told us they completely avoid the REST architectural style. Comparing these numbers to our 2017 Integration Survey, a startling trend appears. In last year's survey, 68% of respondents reported using REST whenever possible, 6% told us they use REST whenever possible, with a few exceptions, and 20% said they had no REST policy or preference. Thus, among respondents, it appears that the trend in the RESTful API space is to move more towards a model of using RESTful APIs for most, but not all, situations.

Even with the trend towards a more moderate use of RESTful APIs, a majority of respondents still build and use RESTful APIs. When we asked, "How do you version your REST APIs?", we saw a rather even split among several different answers choices. The following responses proved the most popular:

ROUGHLY SPEAKING, HOW MUCH TIME DO YOU SPEND ON INTEGRATION?



HOW DOES YOUR ORGANIZATION BUILD AND USE REST APIS?

- 31% said they version in the endpoint URI without aliases.
- 25% version in the endpoint URI with backward compatible aliases.
- 23% do no perform explicit versioning.
- 21% use an API gateway with routing.
- 20% using versioning in the codebase.

No matter the ways in which respondents choose to version their APIs over time, there's consistently only one form of supplying an API with data that's widely popular. In this year's survey, 93% of respondents reported using JSON as their data serialization and interchange format. We saw the same massive majority of JSON users in last year's survey, with 97% of respondents to the 2017 Integration Survey reporting to use JSON for data serialization and interchange. Unlike JSON, the popularity of the other two important forms of data transfer, XML and YAML, has been in constant flux. In 2017, 93% of respondents used XML; this year, only 61% reported using XML. What's more, in our three previous Integration Guides, we'd seen a steady growth among YAML users, going from 34% in 2015 to 39% in 2016 to 46% in 2017. But this year, much like with XML, we saw a startling and precipitous drop, with only 23% of respondents claiming to use YAML for data serialization and interchange.

The reasons for this significant drop in XML users could lie in the increasing popularity of RESTful APIs noted above. According to the article, "Choosing JSON Over XML," "RESTful APIs depend on easy, reliable, and fast data exchanges. JSON fits the bill for each of these attributes, while XML is struggling to keep up." The decline in YAML, however, cannot be explained in these terms. YAML itself is a superset of JSON, and thus reads and performs very similar to its highly popular cousin. Some factors in this year's decline in YAML adoption among our survey respondents could be that the parsers necessary for applications to read YAML files have not yet been built into many languages and because YAML is whitespace dependent, meaning it can be harder for developers to troubleshoot and debug than JSON.

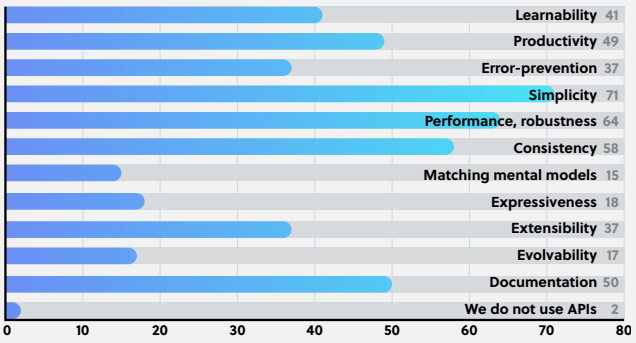
ARCHITECTURE

Now that we've discussed the REST architecture style for APIs and how they're used, let's dive into other popular architectural patterns in the integration field. The two most popular software architecture patterns reported in this year's survey were microservices and MVC patterns. 65% of respondents reported using microservices and 60% reported using MVC patterns. These two constituted the only architectural patterns used by more than 60% of respondents; the third-place pattern, Layered (n-tier) architecture, is used by 45% of survey-takers. Given the popularity of microservices and MVC patterns as compared to other architectures, we'll use these two patterns as a means of exploring how respondents build and interact with various components of software architecture during integration projects.

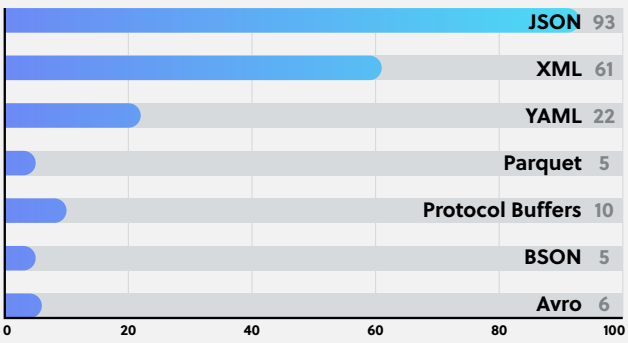
When we asked respondents what integration topologies they use, 62% reported using point-to-point, 51% use message bus, 45% use service bus, and only 20% use hub-and-spoke. When we compare these numbers to the two main architectural patterns, microservices and MVC, something interesting happens. While service bus and hub-and-spoke remain the less popular of the options for users of both patterns, message bus and point-to-point topologies' adoption rates among respondents proved more variable. Among microservices users, 62% use message bus topologies and 60% use point-to-point topologies in their integrations. Among MVC users, however, 67% reported working with point-to-point topologies and 54% use message bus topologies. Not only did the positioning in terms of most adopted topology change between mircoservices and MVC users, but message bus topologies proved 8% less popular among MVC users than microservices users.

When asked about quality attributes they keep in mind when deciding on an architectural pattern for their project, respondents reported a wide range of options. Among all respondents, 74% look for scalability and performance, 69% take agility into consideration, and 54% look at ease of deployment. While there were several more answers reported by respondents — including loose coupling (52%) and testability (51%) — scalability, performance, and agility proved the most popular quality attributes to consider. When we compare these quality attributes to the numbers for microservices and MVC users, we see a similar pattern to the one outlined in the previous paragraph. Among respondents who work with microservices, 79% identified scalability as an important quality attribute, 76% reported performance, and 73% said agility. 77% of respondents who work with MVC patterns, however, reported performance as an important attribute, 73% selected scalability, and 71% told us agility. Again, we see the top answer trading places depending on if respondents work with microservices or MVC architecture patterns.

WHICH OF THESE API QUALITY ATTRIBUTES ARE MOST IMPORTANT WHEN YOU ARE USING AN API?



WHICH OF THE FOLLOWING DATA SERIALIZATION AND INTERCHANGE FORMATS DO YOU USE?



INDUSTRY'S FASTEST DRIVERS

Know SQL?

Connect to 100s of Apps, Databases, & SaaS APIs though SQL

SQL is the language of data, and our state-of-the-art drivers let you leverage the full power of SQL to connect with hundreds of applications, databases, and APIs.

Through standards-compliant driver technologies like ODBC, JDBC, ADO.NET, & OData, you can read, write, and update live data, from any data source, using standard SQL queries.

Features:

- Real-Time Access to Data
- Rich Metadata Discovery
- Robust SQL-92 Support
- High-Performance Data Connectivity

ODBC JDBC ADO.NET SSIS CLOUD BIZTALK EXCEL MOBILE

Work With Relational Data, Not Complex APIs or Services

Whether you are a developer using ADO.NET, JDBC, OData, or MySQL, a systems integrator working with SQL Server or Biztalk, or even an information worker familiar with ODBC or Excel — our products give you bi-directional access to live data through easy-to-use technologies that you are already familiar with. If you can connect to a database, then you will already know how to connect to Salesforce, SAP, SharePoint, Dynamics CRM, Google Apps, QuickBooks, Sage, NetSuite, and much more!

Visit **www.cdata.com** to learn more and download FREE Trials.

SDKs and World Class Developer Experience

QUICK VIEW

01. Determine the programming languages you will support.

02. Develop a strategy to support the programming languages you don't know.

03. Create a GitHub repository to support each SDK.

04. Support, collaborate, and manage your open source community.

BY ELMER THOMAS
DEVELOPER EXPERIENCE ENGINEER, **SENDGRID**

Developing an SDK for your API can significantly increase the developer experience by providing native support for a developer's preferred programming environment. This article will describe some best practices, strategies, and processes to help your API provide a world class developer experience.

CHOOSING WHAT LANGUAGES TO SUPPORT

When launching support for multiple programming languages, build the SDKs out one language at a time. Ideally, interview a sample of your customers and find out what languages and platforms they prefer. If that's not feasible, use one of the surveys released by GitHub or StackOverflow and choose one of the top three to start. Continue your research on GitHub, searching for any open source code that already utilizes your API.

You can also try creating some basic documentation with page view tracking that demonstrates how to interact with your API using basic calls in several popular languages. Put each programming language example on its own page and observe the page views to help inform your priority.

As you release the SDKs, include a user agent in the header to record usage data, which can then be used to help calculate your RICE score (a method of prioritization), as explained in further detail in the Prioritization section.

SUPPORTING PROGRAMMING LANGUAGES YOU DON'T KNOW

In this case, develop a one to two-week learning sprint before diving into the development of the SDK. Start by consuming high-level content, such as the philosophy behind the language and the common use cases.

Discover the key people behind each programming language's community and consume their content. Reach out to people in your network as well.

Build an HTTP client that consumes your API using the native methods of that programming language to cement your knowledge. Finally, take some time to explore a few quality SDKs from top-tier API providers.

CREATING THE GITHUB REPOSITORY

At a minimum, your SDK should provide documentation and tested code hosted on a platform such as GitHub. Often, your documentation will live in at least two locations: GitHub and your official website. Keeping these documents in sync is a challenge therefore, utilize the OpenAPI specification as the source of truth for both your documentation and code.

While you can write the OpenAPI specification by hand, do not take that approach as it's generally tedious and prone to error. Services such as StopLight.io and Apiary.io allow you to define your API with a pleasant UI that's well-suited for collaboration and then export the results as OpenAPI-compliant JSON.

Many open source SDKs contain, at a minimum, the following elements: the code, LICENSE.txt, README.md, CONTRIBUTING.md, CODE_OF_CONDUCT.md, USE_CASES.md, CHANGELOG.md, TROUBLESHOOTING.md, and a set of labels applied to each

issue or pull request. For an example repo that embodies these elements, please see this repository.

But first, take a moment to set up the repository metadata. Near the top, you can add a description, website, and topic labels. As you build out your SDK, keep a journal to document your process. As part of this journal, create checklists for anything you need to do more than once. Make it a habit to utilize the checklist each time so that you can continue refinement. When you feel like the checklist is working smoothly, look toward automation and process documentation.

THE CODE

SDKs consist of two layers: the interface to the API and the helper code that helps execute specific use cases. The former can be automated, and the latter is key to the developer experience.

The helper code should be lovingly crafted, preferably in collaboration with your users. Start with a usage guide published in your GitHub repository that demonstrates how the SDK would work when completed. Then, invite your open source community into a conversation around that document, encouraging comments and pull requests. I found that specifically tagging people in the comments works best to initiate the conversation.

For testing, Stoplight.io provides a free service, Prism, that turns any OpenAPI specification JSON into a live mock server to run your integration tests against. My preferred workflow is to spin up an instance of Prism in a Docker container for local testing and in Travis CI. Utilizing this strategy allows you to skip learning and implementing the specific mocking features for each programming language you support.

LICENSE.TXT

Determine the license that is most appropriate for your situation and put in a LICENSE.txt file in the root directory. Utilize ChooseALicense.com to help choose an appropriate license for your project. For a complete reference of open source licenses, refer to OpenSource.org/licenses.

README.MD

A README is perhaps the most important document in your repository as it's usually the first impression someone has of your project. At a minimum, your README should contain the prerequisites, installation instructions, and a quick start example with the goal of getting your user to their first API call as quickly as possible.

Additionally, I recommend that you include the following elements in your README:

- Summary
- Logo/branding
- Build status badge
- Table of contents
- Announcements
- Quick start example
- Roadmap
- Links to
  - API documentation
  - Use case documentation
  - Contributing guide
  - Troubleshooting guide
- About section

CONTRIBUTING.MD

This document should be designed such that anyone can easily contribute to your repository, from fixing a typo to implementing a brand-new feature. Cover the following topics in your contributing guide:

- CLA/CCLA/DCO requirements
- Roadmaps & milestones
- Submitting a feature request
- Submitting a bug report
- Suggesting improvements to the codebase
- Understanding the codebase (a high-level overview of the code)
- How to install a developer environment and run the tests
- Style guidelines and naming conventions
- How to create a pull request
- How to conduct a code review

CODE_OF_CONDUCT.MD

The code of conduct should set the tone for your community while describing a reporting mechanism for violations with a clear description of the consequences. Check out GitHub's built-in code of conduct functionality for inspiration.

USE_CASES.MD

Your "getting started" documentation may be THE most important key to a world class developer experience. Here, you need to define the most important jobs to be done for your customers.

The goal is to get them from reading this document to making their first API call as quickly and easily as possible.

CHANGELOG.MD

In addition to leveraging the release functionality built into GitHub, create a CHANGELOG.md file that minimally includes the semver version, release date, and the type of changes (e.g. fixes, features, etc.). I like to link each entry to the original pull request, issue, and the contributor's GitHub profile along with a note of thanks.

TROUBLESHOOTING.MD

First, try to think of common issues that your users may experience and document them here. As issues are reported, document any workarounds or gotchas here. Then, use this document to help you drive an improved developer experience over time by examining how you can remove each issue from this document through improving the SDK's functionality.

LABELS

GitHub labels can be used to communicate the status of an issue or pull request and to trigger automations. Here are some labels to use as a starting point:

- Difficulty
  - difficulty: easy
  - difficulty: medium
  - difficulty: hard
  - difficulty: very hard
  - difficulty: unknown or n/a
- Type
  - type: bug
  - type: enhancement
  - type: docs update
  - type: question or discussion
- Status
  - status: duplicate
  - status: help wanted
  - status: invalid
  - status: wontfix
  - status: cla or dco not signed
  - status: code review request
  - status: ready for merge
  - status: ready for deploy
  - status: work in progress
  - status: waiting for feedback
- Misc
  - hacktoberfest
  - up-for-grabs
  - up for grabs
  - help wanted
  - spam
  - good first issue

Then, you can apply the following process upon intaking each issue or pull request:

1. Assign a Difficulty
2. Assign a Type
3. Assign a Status
4. Optionally assign a Misc. label

SUPPORT, COLLABORATION, AND MANAGEMENT

One key to scaling the number of people your project can support across many programming languages is to collaborate with your open source community. Serve them well and they will return the favor.

ENCOURAGING COLLABORATION

Why would someone donate their time to your SDK for free? Whatever the reason, you should be rolling out the red carpet.

Be respectful of their time and personally respond to every question and comment. Sure, that part is tough to scale, but if you expect your community to support you, provide them with as many reasons as you can. Simple, timely communication is the baseline.

As you develop relationships with your community, take special note of your top contributors and actively seek their advice. If possible, speak to them on the phone, through video chat, or ideally in person.

Chances are your company relies on open source for many projects, so it's natural to encourage giving back. Utilize internal communication platforms to seek out open source advocates and form relationships. Leverage internal newsletters to create specific calls to action. For example, perhaps one of your SDKs could use some help with code reviews, so send out an email with direct links to where help is needed along with detailed instructions on the process.

ROADMAPS

If you take the time to expose your roadmap, you can learn whether it resonates with your community while also offering an opportunity for potential contributors. You can either use the built-in milestone and/or project feature in GitHub, create a simple ROADMAP.md file, or create groups of issues tagged with roadmap labels.

CODE REVIEWS

Performing code reviews can be time-consuming, especially if the code is written in a programming language that is not part of your expertise. By defining the process for performing a code review and codifying it in your .github templates and CONTRIBUTING.md files, you can make it easier to solicit help from your community.

ISSUES AND PULL REQUESTS

Issues can be used to ask questions, report bugs, and request features. Encourage all three of these scenarios as well as commenting on issues. Document the process for each of these scenarios in your .github templates and in your CONTRIBUTING.md files.

A pull request can be in response to an issue or spontaneously submitted. These are both fantastic, but additional care must be taken on the latter since it's unwise to simply accept a pull request that does not have a high RICE score (see the Prioritization section below for further details). Have a strategy for saying "no" without severing the relationship.

When a new issue or pull request comes in, utilize your labeling process and apply automation to appropriately prioritize. Respond to incoming issues and pull requests at least once per day manually, even if it's just to say that you received the issue and will follow-up at a later date. You may also want to employ an automated message that guides the submitter to next steps and expectations.

Pull requests are the ultimate expression of developer love and should be treated as such. Whenever possible, prioritize these.

PRIORITIZATION

Utilize the RICE prioritization method: Reach, Impact, Confidence, and Effort. The idea is to assign a numerical score to each incoming issue and pull request, computed using this formula: (R*I*C)/E. This score can be used to determine what you should work on next.

- Reach can be computed by estimating how many of your users this issue or pull request would affect.
- Impact is subjective, keep it simple with categories like large, medium, and small.
- Confidence is your idea on how much confidence you have in the reach impact and effort estimations.
- Effort is how difficult you think it will be to complete.

THE BACKLOG

I like to keep two backlogs for managing SDKs: one for projects and the other for issues and pull requests, assigning portions of the day/week to each of the two backlogs. To start, begin with a spreadsheet and, once the system is working efficiently, converting the system to a database driven application.

RELEASE MANAGEMENT

To make the developer experience seamless, deploy your SDK to the relevant package managers. Depending on the frequency of contributions, either release on-demand or batch releases at some predetermined cadence. Define both cases because, at a minimum, you will need to utilize on-demand releases for bug fixes.

Below is an example release checklist. (Note: you need to fill in the details for each bullet point with enough detail that you would be able to outsource or automate the entire process):

1. Ensure all tests pass
2. Ensure code review is complete
3. Update CHANGELOG.md
4. Update README.md
5. Update CONTRIBUTING.md
6. Update USE_CASES.md
7. Upload to package manager
8. Update releases section in GitHub
9. Perform post-release smoke test
10. Announce externally

Adopt a versioning standard and consistently apply it to every release. Be very careful with this as it's a trust indicator, especially when it comes to breaking changes. Follow the semver standard and clearly define what patch, minor, and major version bumps mean.

THANKS!

To continue the conversation, please reach out on Twitter @thinkingserious! Happy Hacking!

ELMER THOMAS completed a B.S. in Computer Engineering and a M.S. in Electrical Engineering at the University of California, Riverside. His focus was on Control Systems, specifically GPS navigation systems. He currently serves as the Developer Experience Engineer at SendGrid, leading, developing and managing SendGrid's open source community, which includes over 24 active projects across 7 programming languages. More information about Elmer can be found at his blog, ThinkingSerious.com.

EVERYTHING IS GOING TO BE 200 OK®

Runscope

Runscope's API monitoring provides uptime, performance, and data validation for any API.

Easily create scheduled API monitors from 16 different locations across the globe.

"Runscope opens our eyes to a broad range of API problems, including issues so small they would otherwise be undetected."

- Product Manager, SendGrid

Start Your Free Trial Today at runscope.com/dzone

Solve API Issues Faster with API Monitoring

Nowadays, with the popularity of microservices and serverless architectures, applications that rely on dozens or hundreds of APIs are increasingly more common. The rise of third-party APIs that can do one function of an application really well such as payment (Stripe), or email (SendGrid), or messaging (Twilio), have also boosted the productivity of developer's team immensely. But with applications relying more and more on APIs, what happens when they break?

Broken APIs can lead to broken applications, which leads to poor customer experience and lost revenue, and can affect not only a development team, but also support, sales, IT, and other departments. A good testing process can help mitigate some of those issues, but when it comes to APIs, it is crucial to have a solid monitoring strategy.

Ask any API developer what their biggest challenges are when working with APIs, and visibility will always be in the top three. API monitoring can give development teams the insight they need into what is happening behind the scenes and decrease the overall time to resolve issues significantly.

A proactive API monitoring approach will ensure that:

- Your internal, external, and 3rd-party APIs are up and available.
- Your APIs performance is fast, and you can stay ahead of problems that could cause outages.
- Your APIs data is always returning the correct format, structure, and content that you expect.

API management solutions will get you 90% of the way to designing, creating, and publishing a great API. API monitoring will help you and your team cover the last 10%, and it can mean the difference between a failed application and an API that will truly delight your users.

WRITTEN BY BRYAN WHITMARSH
SR. DIRECTOR OF PRODUCT MANAGEMENT, RUNSCOPE

PARTNER SPOTLIGHT

Runscope

Everything is going to be 200 OK®

Runscope

CATEGORY
API Monitoring

RELEASE SCHEDULE
Continuous, SaaS delivery

OPEN SOURCE?
No

CASE STUDY

SendGrid is a leader in customer communications, delivering transactional and marketing email through one reliable cloud-based platform. More than 74,000 businesses globally use SendGrid to send over 45 billion emails per month, including Airbnb, Booking.com, Intuit, Spotify, and Uber.

SendGrid was leveraging scripts and integrations tests for each API endpoint, but they quickly realized that wasn't scalable. Once they signed up for Runscope, they were able to quickly go from idea to production, and fully monitor their API endpoints individually, and also cover complex workflows. "Runscope allows us to continuously monitor the health of our services from the customer perspective. The ability to know where slowdowns or errors are occurring aids us in troubleshooting and getting in front of potential issues," said Acosta.

Today, SendGrid monitors their APIs by the minute, and from multiple locations. That way, they can ensure a great performance for their users across the globe and be notified of any issues before their customers do.

STRENGTHS

1. Intuitive UI to easily create API monitors
2. Uptime and performance monitoring
3. Verify APIs are returning correct data
4. Integrate with Slack, PagerDuty, and other popular tools
5. Monitor internal APIs with on-premises agent

NOTABLE USERS

- Twilio
- Microsoft
- Adobe
- Okta
- Tesco

WEBSITE runscope.com **TWITTER** @runscope **BLOG** blog.runscope.com

The Role of API Gateways in API Security

BY GUY LEVIN
CTO, **RESTCASE**

QUICK VIEW

01. API security experts warn that security is often an afterthought, meaning those APIs may be likely to leak sensitive data or be a target for attackers.

02. Without a gateway, all microservices must be exposed to the "external world," making the attack surface larger than if you hide internal microservices not directly used by the client apps.

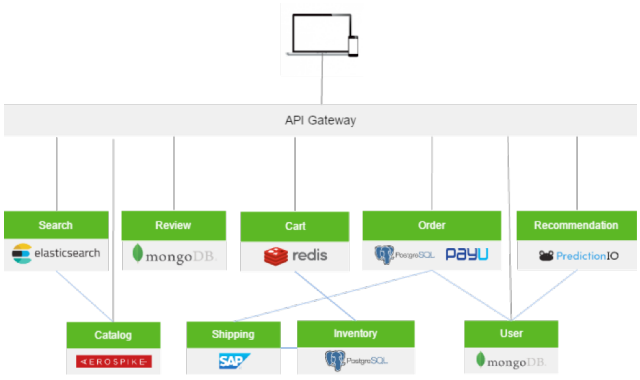
03. An API Gateway requires additional development cost and future maintenance if it includes custom logic and data aggregation.

When switching from a monolith application to microservices, the behavior from the client side cannot be same as it was, where the client had one entry point to the application.

Now, when working with microservices, the client has to deal with all the complexity that comes from a microservices architecture, like aggregating the data from various services, maintaining several endpoints, the increased chattiness of client and server, and having separate authentication for each service.

Client dependency on microservices directly makes it difficult to refactor the services, as well. An intuitive way to do this is to hide these services behind a new service layer and provide APIs that are tailored to each client.

This aggregator service layer is also known as an API Gateway, and it is a common way to tackle this problem.



API Gateway-based microservices architecture pattern.

All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice.

A typical API Gateway includes:

- Security (authentication and potentially authorization)
- Managing access quotas and throttling
- Caching (proxy statements and cache)
- API composition and processing
- Routing (possibly processing) to an "internal" API
- API health monitoring (performance monitoring)
- Versioning (possibly automation)

API GATEWAY ADVANTAGES

- Implemented in a single place
- Simplifies the API source code itself, since these concerns are externalized
- Provides a central and unique view of the API and therefore be more likely to allow a consistent policy

API GATEWAY DRAWBACKS

- Possible single point of failure or bottleneck
- Risk of complexity since all the API rules are in one place
- Risk of lock-in, and migration may not be simple

API GROWTH CREATES OPPORTUNITIES, VULNERABILITIES

To get a grasp on the runaway growth of APIs, one needs only to

look at statistics from ProgrammableWeb, which has been tracking publicly exposed APIs since 2005. At that time, there were only around 100 APIs listed; today, there are more than 10,000 publicly known APIs.

That growth is increasingly underpinning an economy that is reliant on treasure troves of user data. Salesforce.com reportedly generates more than 50% of its $3 billion in annual revenue through its APIs, and nearly 90% of Expedia's $2 billion in annual revenue.

Companies generate API revenue by metering access to APIs and the resources behind them in a variety of ways. For instance, Twitter, Facebook, and others provide ad-based APIs that allow for targeted advertisements based on reporting and analytics, but ad agencies and other brands must pay for access to those APIs.

THE API GATEWAY'S ROLE IN SECURITY

IDENTITY AND ACCESS

Access control is the number-one security driver for API Gateway technology, serving as a governor of sorts so an organization can manage who can access an API and establish rules around how data requests are handled.

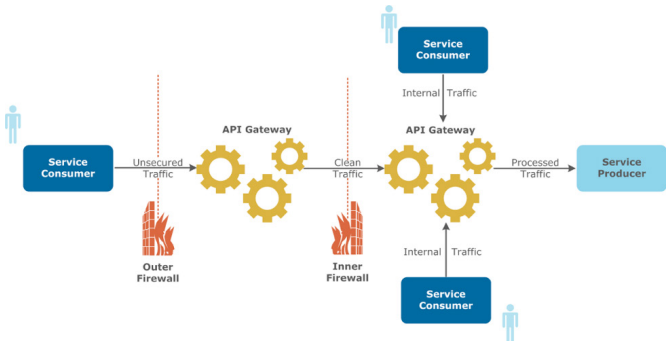
Cheshire said access control almost always extends to establish other policies, including rate limits on API calls from certain sources, or even payment requirements for accessing all or certain resources through an API.

When all traffic is routed through a gateway, IT security experts feel more confident that they have their finger on the pulse of an organization.

An API Gateway's access control capabilities usually start with authentication mechanisms to determine the actual source of any API calls. Currently, the most popular gateway is OAuth, which acts as an intermediary for accessing web-based resources without exposing a password to the service, with key-based authentication reserved for instances in which the business can afford to lose the data because it's difficult to guarantee complete secrecy of the keys.

MESSAGE SECURITY

Gateways are a great way to route all API transactions through a single channel for evaluating, transforming, and securing messages across an organization. When all traffic is routed through a gateway, IT security experts feel more confident that they have their finger on the pulse of an organization.



API Gateway can introduce message security between the internal services making the internal services to be more secure and the messages going back and forth between the services encrypted.

Ignoring proper authentication — even if transport layer encryption (TLS) is used — can cause problems. With a valid mobile number in an API request, for instance, any person could get personal email addresses and device identification data. Industry-standard strong authentication and authorization mechanisms like OAuth/OpenID Connect, in conjunction with TLS, are critical.

THREAT PROTECTION

Without threat protection, the API Gateway, its APIs, and the native services of the integration server are basically insecure. That means potential hackers, malware, or any anonymous outsiders could easily attempt to propagate a series of attacks such as DDoS or SQL injection.

APIs are the gateways for enterprises to digitally connect with the world. Unfortunately, there are malicious users who aim to gain access to backend systems by injecting unintended commands or expressions to drop, delete, update, and even create arbitrary data available to APIs.

In October 2014, for example, Drupal announced a SQL injection vulnerability that granted attackers access to databases, code, and file directories. The attack was so severe that attackers may have copied all data out of clients' sites. There are many types of injection threats, but the most common are SQL Injection, RegEx Injection, and XML Injection. More than once, we have seen APIs go live without threat protection —it's not uncommon.

LOGGING

Many API developers become comfortable using 200 for all success requests, 404 for all failures, 500 for some internal server errors, and, in some extreme cases, 200 with a failure message in the body, on top of a detailed stack trace. A stack trace can potentially become an information leak to a malicious user when it reveals underlying design or architecture implementations in the form of package names, class names, framework names, versions, server names, and SQL queries.

It's a good practice to return a "balanced" error object, with the right HTTP status code, minimum required error messages, and no stack trace during error conditions. This will improve error handling and protect API implementation details from an attacker.

The API Gateway can be used to transform backend error messages into standardized messages so that all error messages look similar; this also eliminates exposing the backend code structure.

WHITELISTS AND WHITELIST-ALLOWABLE METHODS

Considering API traffic at the IP address level, there should be a known list of devices, servers, networks, and client IP addresses. Depending on the tightness of the network, this list will vary in size.

It is common with RESTful services to allow multiple methods to access a given URL for different operations on that entity. For example, a GET request might read the entity, while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity.

It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (for example, a 403 Forbidden).

INPUT VALIDATIONS

Taking advantage of loose input validations allowing a hacker to find the gaps in a system. Using existing inputs, an attacker will explore what is accepted or rejected and push what is possible until they find a way into an API and break down the system's integrity.

Here are the most common input validations.

MESSAGE SIZE

It is good to have message size limitations. If you know with 100% certainty that you are not going to receive large messages (for example, more than 2MB), why not filter them out?

SQL INJECTION

SQL injection protection allows you to block requests that could possibly cause an SQL injection attack.

JSON THREAT PROTECTION

JavaScript Object Notation (JSON) is vulnerable to content-level attacks. Such attacks attempt to use huge JSON files to overwhelm the parser and eventually crash the service.

XML THREAT PROTECTION

Malicious attacks on XML applications typically involve large, recursive payloads, XPath/XSLT or SQL injections, and CData to overwhelm the parser and eventually crash the service.

For more info about input validations, please visit here.

RATE LIMITING

Requiring authentication for all API users, and the logging of all API calls made allow API providers to limit the rate of consumption for all API users. Many API Gateways allow you to put caps on the number of API calls that can be made for any single API resource, dictating consumption by the second, minute, day, or other relevant constraint.

API GATEWAYS: OPEN SOURCE

Here are some of the products worth checking out:

- Tyk
- WSO2 API Manager
- Kong Community Edition

CONCLUSION

When talking about API security, we must understand that security is the number-one concern of companies, organizations, institutions, and government agencies considering investing more resources into their API infrastructure, as well as companies that are ramping up their existing efforts. At the same time, it is also the most deficient area when it comes to investment in API infrastructure by existing API providers. Many companies are building APIs as products on their own, deploying web, mobile, IoT, and other applications, but rarely stopping to properly secure things at each step along the way, but API Gateways are one of the most popular and efficient solutions for the many security problems you'll face.

GUY LEVIN is the CTO of RestCase, a company that develops a REST API Development Platform that helps software companies and developers speed up development of RESTful services, minimizing time-to-market and improving quality. Guy is a full stack engineer, DBA, and architect focused on distributed and cloud systems. He has over 20 years' experience in a variety of sectors including medical and healthcare, cyber security, and fintech.

diving deeper

INTO **API MANAGEMENT**

twitter

 **@agoncal**

 **@unix_root**

 **@inadarei**

 **@esther_gross**

 **@kinlane**

 **@mandywhaley**

 **@jeremiahg**

 **@tedepstein**

 **@apihandyman**

 **@rossmason**

resources

The Basics of API Management

Learn from the API Evangelist about API management: perhaps the most mature aspect of the API economy.

5 Rules for API Management

Learn about the five rules for API management, which involve design, documentation, analytics, universal access, and uptime.

API Management 101

In the first of a series on API management from APIacademy, learn about the nuts and bolts of API management.

zones

Integration dzone.com/integration

The Integration Zone focuses on communication architectures, message brokers, enterprise applications, ESBs, integration protocols, web services, service-oriented architecture (SOA), message-oriented middleware (MOM), and API management.

Microservices dzone.com/microservices

The Microservices Zone will take you through breaking down the monolith step-by-step and designing microservices architecture from scratch. It covers everything from scalability to patterns and anti-patterns. It digs deeper than just containers to give you practical applications and business use cases.

Cloud dzone.com/cloud

The Cloud Zone covers the host of providers and utilities that make cloud computing possible and push the limits (and savings) with which we can deploy, store, and host applications in a flexible, elastic manner. The Cloud Zone focuses on PaaS, infrastructures, security, scalability, and hosting servers.

refcardz

RESTful API Lifecycle Management

In this Refcard, familiarize yourself with the benefits of a managed API lifecycle and walk through specific examples of using RAML to design your API.

REST API Security

API security is the single biggest challenge organizations want to see solved in the years ahead. Download this Refcard to gain a better understanding of REST APIs, authentication types, and other aspects of security.

Cloud Capacity Management

With the increasing use of cloud environments in IT, it's key to ensure you're managing and optimizing your cloud resources as you would with on-premise resources. This Refcard dives into what it means to extend capacity management to the cloud, how it differs from traditional on-premises capacity management, and how to apply it in specific use cases.

courses

Managing, Monitoring, and Subscribing APIs With IBM API Connect V5

In this 16-hour intermediate course, learn how to manage provider organizations and developer organizations, how to analyze the API event record and statistics, and more.

REST API Design, Development, and Management

Learn about REST API concepts, get hands-on API management practices, and learn about some API design and security best practices.

SAP API Management Training

Coordinated by some of the best industry experts, this SAP API Management tutorial offers professional insight over modules.

Microservices & Microservice Architectures

Build your application architecture with microservices and APIs for agility, scale and security

CA Technologies provides the proven platform for a scalable, secure microservices solution for the enterprise.

Explore https://www.ca.com/us/why-ca/microservices-architecture.html

ca technologies

Accelerate microservices and API development with tools from CA Technologies.

Faced with the app economy, many enterprises must rebuild applications that need to quickly adapt to changing needs; and the traditional way of rolling out (and supporting) large applications just isn't sufficient. Today's enterprise architects and VPs of applications are wondering:

- How can I deploy and release modern applications in days or weeks, not months or years – and minimize downtime on app updates?
- How can I leverage multiple development teams on different language platforms to build those modern applications?
- How can I scale applications as needs change, while minimizing infrastructure costs to accommodate that scaling?

These challenges stem from an increased focus on agility and scale for building modern applications — and traditional application development methodology cannot support this environment. CA Technologies has expanded full lifecycle API management to include microservices — an integration enabling the best of breed to work together to provide the platform for modern architectures and a secure environment for agility and scale. CA enables enterprises to use best practices and industry – leading technology to accelerate and make the process of architecture modernization more practical.

Today's DevOps and agile-loving enterprises are striving for fast changes and quick deployments. To these companies, the microservices architecture is a boon, but not a silver bullet. Organizations can enable smaller development teams with more autonomy and agility, and as a result, the business will notice IT is more in tune with their changing demands. IT will need to align its API strategy with the microservices that developers produce. Securing those microservices should be of the utmost importance; leveraging API Gateways in this context will benefit IT. And always remember, that if you're looking for speed and scale, safety is equally important — and a strong management component is a must.

WHY ARE MICROSERVICES SO IMPORTANT?

Every digital enterprise trying to thrive in the digital economy is aspiring for two things: speed and scale. If a company's need to get to market faster is critical, it's equally important to be able to scale up appropriately to support increasing customer demand. But the key mantra here is: speed and safety at scale. You can only succeed when you attain speed and scale without losing safety. Agile and DevOps models support decentralized and distributed ownership of software assets and promote faster turnaround of changes and quick deployment. However, to intelligently break down complex, monolithic applications into autonomous units, you need a design strategy, namely, microservices.

By breaking your huge application into microservices, you're enabling your development team to be nimbler with updates and autonomous deployments. This removes dependencies to create large and complex builds, and it eliminates the need for over-sophisticated architectures to step up scalability to meet volume demands.

WRITTEN BY BILL OAKES, CISSP
DIR. OF PRODUCT MARKETING FOR API MANAGEMENT, **CA TECHNOLOGIES**

CA Technologies provides the proven platform for a scalable, secure microservices solution for the enterprise.

PRODUCT STRENGTHS

- Centralized security enforcement for authentication, authorization, and threat protection
- Routing and mediation to protected resources across various protocols
- Service-level management for enforcing business-level rate limits and quotas
- Service orchestration for reducing service invocations
- Service façades for exposing application-specific interfaces from monolithic back ends

WEBSITE bit.ly/2AnOPMs **TWITTER** @CAApi **BLOG** bit.ly/2nCZXz8

Systems Integration Before REST

QUICK VIEW

01. System integration is a practice that's always evolving.

02. We went from low-level integration of function calls all the way up to high-level integration using language-agnostic methodologies and technologies.

03. All types of integration reviewed in this article are still valid and in-use, it all depends on the use case.

BY FERNANDO DOGLIO
TECHNICAL MANAGER, **GLOBANT**

IN THE BEGINNING THERE WAS NOTHING

If you think about it, standards and protocols start popping up after the task they're meant to simplify has been around for a while and no two groups are doing it the same way. This holds true for hardware, if you look up the history of how computers came to be a household item, you'll see how it was every company for itself in the beginning. This holds true for software as well, a recent example being how mobile development got standardized and you can now even create a single application that will work on all major operating systems (it wasn't long ago when you had to use different technologies for different models of devices from the same company). So, in the beginning, when the need for distributed computing appeared and the different systems needed to start communicating with each other, the first solutions weren't exactly open.

In fact, in the '70s (and early '80s), one of the first recorded system integration tools was called "EDI" (Electronic Data Interchange). It was developed to allow companies to exchange valuable documents seamlessly with each other. Now, surprisingly enough (not really), there was no standard that could easily be implemented in any language. Instead, EDI provided a set of software tools that would let you perform the exchange. Another key aspect to EDI is the fact that the documents being exchanged had very specific and well-defined formats they were allowed to be in.

This is not like forcing XML into your message format (like SOAP does), this is a protocol designed to exchange documents, yet these documents could only be transferred if they complied to one of the approved standards (the X12 Document list page in Wikipedia contains the full list of these standards).

Moreover, EDI allows for different transmission mediums, which is something that is not that common. Some of the most used integration protocols these days work on-top of HTTP, but aside from also working with HTTP, EDI also allows for channels like FTP, e-mail, AS1 (networking protocol based on SMTP), and others.

Even though EDI has been around for over 40 years and works by forcing the standard of the documents being transferred, it is used by many governments internally to share documents between organizations. The list of standards is also constantly updated, so new ones get added as soon as the need for them arises.

INTEGRATING SYSTEMS BY REMOTE PROCEDURE CALLS

RPC was developed during the '80s and instead of integrating systems by allowing them to exchange digital documents, it allows distributed systems to integrate with each other by remotely executing procedures (or subroutines) as if it was all a single system.

Tracing the origin of this technology is a bit tricky, since some would say that there are theoretical implementation proposals that date back all the way to the '70s, but the first few practical implementations started appearing in the early '80s. In fact, the term RPC is attributed to Bruce Jay Nelson, an American computer scientist, in 1981.

That being said, RPC has one minor problem, which I would attribute to it being the first attempt at solving a problem that was new at the time: the implementation is language dependent. In fact, some of the first implementations actually required the creation of

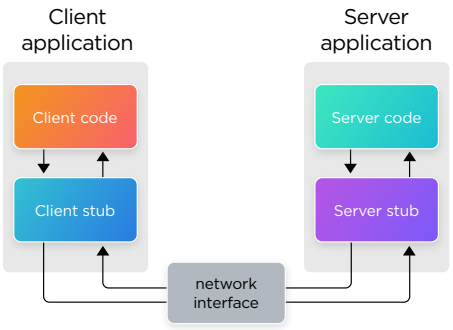
dedicated programming languages, such as Lupine, the first actual practical implementation of RPC, developed at XEROX by Nelson and Andrew Birrel.

The first popular implementation of RPC was Sun's RPC, now known as ONC RPC which was used as the basis for NFS (the Network File System).

HOW DOES IT WORK?

You can distill RPC all the way down to a simple client-server communication protocol, in which the calling code acts as the client, and the executing subroutine acts as the server.

It was standardized by providing a simple way to replicate the interface for the remote procedure. Or, put another way, an Interface Definition Language (or IDL for short) is used to define the interface, and, with generators, you can grab these IDL files and generate your own client and server stubs in the language of your choice.



Simplified explanation of interaction between client and server during RPC communication

The previous diagram provides more details on the different components involved in RPC communication. Although the details might vary from implementation to implementation, the basics of it are:

1. The client application binds with the client stub, which is basically a "fake" instance of the remote procedure trying to be executed (same interface, but not the actual procedure).
2. The client code executes the stub, sending it the required parameters.
3. The client stub will marshal the parameters (which is fancy talk for "serialization") and transmits them to the server stub.
4. The server stub will, in turn, unmarshall the package (which, again, is code for recreating the parameters from the received serialized package).
5. The server stub will execute the server code, passing the received (and now unmarshalled) parameters.

The response from the procedure call will go through the same inverse process (marshalling, transmission over the network, unmarshalling, and final reception by the client code) and land on the client.

One of the main drawbacks of this approach is that it tries to hide the non-locality of the server from the developer but has no way to handle network problems on its own. So, the developers need to write network ahandling code on the client-side, breaking the very illusion this approach initially intended to provide (that there is nothing between client and server).

A STEP IN THE RIGHT DIRECTION

The next iteration in the integration effort came with CORBA.

CORBA was born in the early '90s as an attempt to bridge the gap left by RPC and other similar attempts. Even though they all had succeeded in allowing distributed systems to communicate, they had not been as successful providing heterogeneous ways to integrate systems built using different technologies. Some protocols were good for some languages, some others weren't.

So, the Common Object Request Broker Architecture, as defined by the Object Management Group (OMG for short) was an attempt at providing a language and OS-agnostic way of allowing two CORBA-based systems to interact with each other.

At its core, you can say that CORBA is an object-oriented implementation of RPC that does not focus on the language used to create the system. It works in the same way RPC does, by creating and publishing IDLs of the shared services, although this particular IDL is designed and managed by OMG, and clients need to use them to create their stubs as well as the servers to create their skeletons (which would be the server stub from before). It also offers many other benefits, such as:

1. Heterogenous access: Language and OS freedom is one of the key winning features of CORBA over previous attempts at system integrations.
2. Data-typing: The IDLs allowed developers to map the types between systems and this meant mapping between not-entirely-compatible technologies.
3. Better transfer error handling: CORBA allowed applications to determine if a call had failed due to network problems or due to other issues.
4. Finally, data compression while marshalling the parameters to be sent back and forth was added. This was developed by IONA, Remedy IT, and Telefonica and later added by OMG as part of the standard.

I SPY A LITTLE API

For all the benefits that CORBA provided, once the W3C (the World Wide Web Consortium) released their XML specification, systems integration took a different direction. Suddenly, Microsoft was able to get major IT players, such as IBM, to start adopting their Simple Object Access Protocol (or SOAP for short) which they had created around 1998.

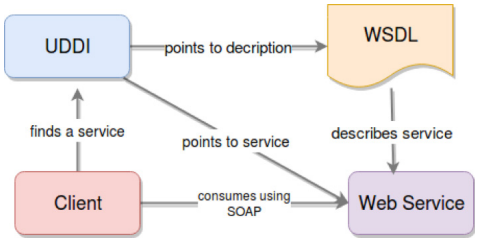
This time, the abstraction layer had been raised again, and now, instead of performing remote method invocation as if they were local, you were actually doing a remote request to an external service. This particular new protocol came with the following benefits over the previous options:

1. It was independent of the programming model used. It wasn't bound to OOP or procedural programming; you could implement it in whatever model you were using.
2. It was extensible. Updates and improvements could be added to the protocol over time in a seamless manner.
3. And it was neutral to the communication layer. SOAP could be implemented over any protocol such as HTTP, SMTP, TCP, and more.

After SOAP was defined, it became the basis for a much larger stack that would be used to define and consume Web Services. This stack was composed of:

- The Web Service Definition Language (WSDL) as the means of defining the interfaces of these services.
- SOAP as the messaging protocol used to transfer data from clients to servers and back.
- The Universal Description Discovery and Integration (UDDI) protocol, which allowed services across the globe to publish themselves in a centralized discovery platform, which allowed clients looking for those services to find them without having to know where they were.

The following diagram shows how the above elements interact with each other:



Simple explanation of the interactions between UDDI, client, and service

SOAP-based services took over the systems integration space for a while, XML was the new standard in town and it came with some much-needed benefits, such as:

- Flexibility. You could use XML for anything you wanted, so your services were both defined by it and transferred their data using it. This simplified development by only requiring users to understand and parse one language.
- Validation. By defining and using XML Schemas, you could verify correctness in your messages using another standard. This simplified the task of creating ad-hoc validations for new services since having a standard validation language led to the creation of validation tools and libraries across all languages.
- Human-readable structure. This was a major benefit at the time. Other solutions would use binary protocols to encode their data, making it impossible for humans to directly read it and validate the format and correctness of it. By having a structure that made its messages human readable it provided developers with a better experience by reducing debugging times.

IN SUMMARY

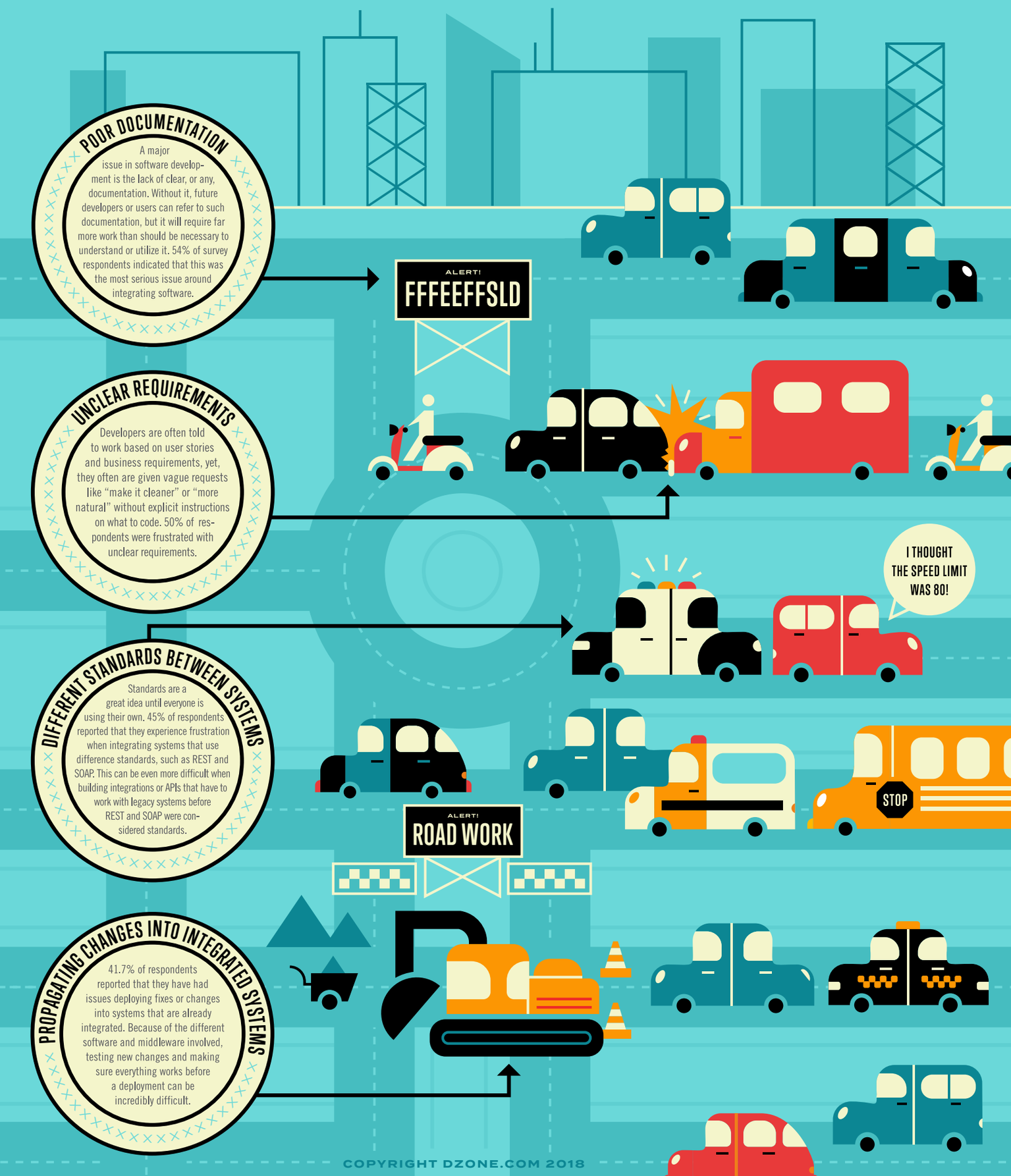
Like I stated in the beginning of this article, systems integration has been around ever since the first two systems needed to talk to each other. The technologies used and the methodologies related to them have evolved over time and new and exciting ways to perform these tasks are developed every year.

The reason this article covers pre-SOAP integration is that just because EDI has been around ever since the '70s, or because REST is "better" than SOAP, it doesn't really mean these technologies aren't used anymore. In fact, they're very much used due to older tech companies not updating systems, either because doing so would be too disruptive to their business or just because it would be too expensive. So, don't be surprised if you happen to find some of these dinosaurs in the wild during your next project!

FERNANDO DOGLIO has been working as a Web Developer for the past 10 years. In that time, he's come to love the web, and has had the opportunity of working with most of the leading technologies at the time, such as PHP, Ruby on Rails, MySQL, Node.js, Angular.js, AJAX, REST APIs, and others. In his spare time, he likes to tinker and learn new things, which is why his GitHub account keeps getting new repos every month. You can read more about him and his work at fernandodoglio.com. When not programming, he is spending time with his family.

Avoiding INTEGRATION FRUSTRATION

Building software integrations doesn't always attract the most headlines, but it's a crucial part of developing enterprise systems. Users may want your software to work with another tool, or you may need to integrate in-house applications with legacy technology. Inevitably, this leads to major roadblocks that keep you from effectively and easily building your own integrations. We asked 712 DZone members what their major sources of integration frustration are:



Introduction to Integration Patterns

BY JOHN VESTER
SR. ARCHITECT, **CLEANSLATE TECHNOLOGY GROUP**
FREELANCE WRITER AND ZONE LEADER, **DZONE**

QUICK VIEW

01. Integration patterns establish the manner in which integrations will be sourced or triggered.

02. Messaging patterns handle the message itself as well as routing and transformation aspects of the content couriered throughout the integration processing and events.

03. System management patterns are used for analysis, administration, and trouble-shooting issues encountered by the system itself.

04 Adoption of established integration patterns can streamline the development process and reduce the ramp-up time for new project team members.

In today's mashup-driven world, the use of integrations to extract, transform, and utilize data is at the top of mind for a majority of software engineers. It is important to understand proven integration patterns which can help streamline the integration process and flows.

INTEGRATION STYLES

When defining an integration between one or more different sources, the "how" question must be answered in order to proceed. In other words, one must determine how the integration will transpire. This is often referred to as the integration style.

REMOTE PROCEDURE INVOCATION

Remote procedure invocation (RPI) is a pioneer in the integration space and was the go-to manner to implement an API in the early days of computing. In this approach, a provider will allow an external process to make requests into a closed application. The external caller has the specifications to make the request and an expectation on what the response will be, but all the logic takes place using a black-box approach. In this case, RPI is the mechanism used to perform some action against the target system.

Consider an application which handles financial transactions. Prior to the popularity of RESTful APIs, the vendor may offer an API to allow transactions to be posted from an external source. This API was implemented using RPI.

The developer would write a program to gather the required information, then connect to the application using RPI. The results of the RPI/API requests were packaged in a response and that information was processed by the calling application.

SHARED DATABASE

The shared database integration style leverages a database for the connectivity between two or more applications. As a result, each application would maintain a connection to a shared database containing the information that will be integrated.

As an example, use of an INSERT statement to a staging table in the database could trigger a stored procedure which would perform business logic — ultimately updating attributes elsewhere in the database for other applications utilizing that same shared database integration.

MESSAGING

The messaging integration style started to gain momentum with service-oriented architecture (SOA) implementations — leveraging an enterprise service bus (ESB) as the foundation for the message itself.

Using the financial transaction example, the custom application may simply place a message on the ESB requesting a certain transaction be posted. That system submits the message and relies on the messaging integration style to handle any remaining tasks

On the financial system side, the message being placed on the bus triggers and event which consumes the message and takes the appropriate action based upon the nature of the message. Based on the message queue used and/or metadata within the message itself, the financial system understands the task that needs to be performed.

When completed, the financial system may place a new message on the bus, which could be consumed by the original system. In this case, it might be related to unique transaction information to append to the original request for audit or validation purposes.

THE MESSAGE CONCEPT

The courier for integrations is largely based around the concept of messaging. This is no different than other technology-driven solutions, as some *thing* is used to pass around information important to the solution at hand. Using RESTful APIs as an example, the courier is often the payload that is passed to a POST request or returned from a GET request.

MESSAGING SYSTEMS

A major benefit the messaging concept is that the asynchronous messages do not require both systems to be online and available at the same time. One system may place a message with an ESB, which could be processed immediately by another system or on a schedule hours later. Either way, both conditions can be handled without impacting the other.

The message system employs channels (or queues) to organize and categorize the information that needs to be integrated. For example, if the source system needs to communicate with a financial system and a HR system, the messages would use different channels for each message type.

MESSAGE ROUTING

The idea of message routing is often implemented in more complex integration scenarios, where a message may be required to route across multiple channels before reaching the target destination.

A message router can assist in this scenario, allowing messages to be submitted to a dedicated component that will analyze the message and employ business logic to determine where to route the message based upon the contents of the message itself.

In the financial transaction example, the source system will simply need to post a transaction. If the corporation maintains multiple financial systems, the source system may not have a detailed understanding of which system handles which transactions. The message router would become the source of the message and would have the proper knowledge to fulfill the delivery of the message to the proper channel.

Message routing goes even deeper and can use a vast array of patterns which assist the routing process. Some common patterns include:

- **Message Filter**: Allows messages to be filtered based upon attributes within the message.
- **Scatter-Gather**: Allows synchronous messages to be sent to multiple sources at the same time.
- **Message Aggregator**: Allows messages from multiple sources to be processed and pushed into a single resulting message, perhaps to process the results from scatter-gather.

MESSAGE TRANSFORMATION

Connecting disparate systems often brings to light that a given response does not match the expected or preferred response from the source system. Message transformation is a mechanism which can perform the necessary conversion of data between the two systems.

Using the financial system example, the source system may wish to send data in JSON, but the financial system is expecting XML. Using message transformation, the incoming JSON data would be analyzed and converted (i.e. transformed) into XML to prepare for processing by a SOAP web service. This is basically the **normalizer** integration pattern in use.

Some established message transformation patterns include:

- **Content Enricher**: Allows metadata to be modified in order to meet the expectations of the target system.
- **Claim Check**: Temporarily streamlines the message in order to remove metadata which is not necessary at that point in time, but available for later processing.
- **Content Filter**: Remove metadata from the message completely, more permanent than the claim check approach noted above.

SYSTEM MANAGEMENT

Building upon the integration styles and the flow and processing of a given message, the management of the integration is the core of the solution.

CONTROL BUS

The control bus pattern is the management tier within the integration system. As one might expect, the control bus employs the same concepts implemented by the integration system.

When the administration layer requires information to report user to the system admin, message data captured by the integration system is utilized to report the status or any known issues that have been encountered.

MESSAGE STORE

Managing any system often requires some level of historical information or metrics. The challenge with metrics examining the messages without impacting the transient nature of the messages themselves. The message store pattern meets this need by sending a duplicate copy of the message to the message store. Once a copy of the message is stored within the message store, the necessary metrics can be maintained and passed to the control bus for processing and reporting.

SMART PROXY

Messages typically flow through to a fixed output channel. However, there are cases where a component needs to post reply messages back to a channel specified in the original request. When this need arises, the smart proxy pattern can be employed.

The smart proxy includes logic to intercept messages in order to capture the return address specified by the sender. Once processing is finished, the smart proxy replaces the fixed output channel destination with the address captured when the original request was received.

CONCLUSION

Maintaining an understanding of the integration styles, message concepts and the system management patterns can help guide integration developers toward employing practices that translate across any integration project regardless of the industry. Doing so will reduce the ramp-up time as additional resources support and maintain existing integration projects.

JOHN VESTER is an IT professional with 25+ years expertise in application development, project management, enterprise integration, and team management. Currently focusing on enterprise architecture/application design/Continuous Delivery, utilizing object-oriented programming languages and frameworks. Prior expertise building Java-based APIs against React and Angular client frameworks. Additional experience using both C# and J2EE.

The Role of APIs in a Microservices Architecture

BY NUWAN DIAS
DIRECTOR, **WSO2**

QUICK VIEW

01. APIs help bring a sense of order into the chaotic nature of a microservices architecture.

02. Managed exposure of microservices help improve developer efficiency by dealing with factors such as uniformity, comprehensiveness, and complex security protocols.

03. Just like with microservices, API Gateways too need to be agile in terms of development, testing, deployment, and scaling. Microgateways play a critical role in achieving that.

To meet the ever-increasing demands of business innovation, organizations are steadily moving towards adopting microservices architectures in their enterprise software. While a microservices architecture provides you with the agility required to develop, maintain, and enhance your software at scale, it doesn't come for free. There are many challenges and hurdles to overcome when applying these architecture patterns and practices. One such challenge is to understand the importance of APIs and API Management in a microservices architecture. To understand the problem better, let us look at the microservices architecture of a retail application.

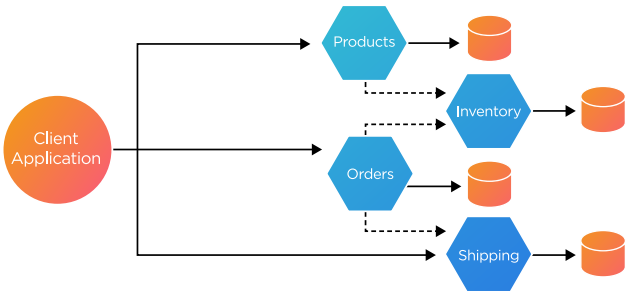


Figure 1 - Microservices Architecture of a Retail Application

Here, we have individual microservices for operations such as browsing the product catalog, placing orders, updating inventories, and shipping items to clients. At first glance, you would realize that there is a lot of chaos in terms of the number of components in action and when it comes to the integration in between these components to facilitate business requirements. Some of the key problems in this architecture are:

1. Too many microservices for the client application to deal with. The nonuniformity across these microservices interfaces, their rawness, and varying authentication schemes would make the consumption of these microservices nontrivial.
2. Policy enforcement on factors such as rate limiting and access control becomes a challenge due to the non-existence of an enforcement point.
3. Business value reporting becomes harder since microservices are more or less good at integrating with systems capable of only performing operational analytics.
4. Inter-microservice point-to-point communication becomes a challenge due to the continuously deployed nature of microservices.

MANAGED EXPOSURE OF THE MICROSERVICES TO CONSUMER APPLICATIONS

UNIFORM EXPOSURE FOR COMPREHENSIVENESS

Each of the individual microservices in Figure 1 could be developed by independent engineering teams. They may have freedom of choice to use their own programming languages, use their own standards, and to release at their convenience. In such situations, it is highly likely each of the microservices would have different interface standards and patterns. This lack of uniformity makes it harder for its consumers to use these services effectively, reducing developer efficiency. Exposing them through a managed infrastructure such as an API Management solution provides an organization with ways of ensuring the APIs exposed for consumer applications are standardized, well documented, and discoverable on developer portals.

API FAÇADE

There are many cases in which the rawness of the microservices should not be exposed to its consumers. For example, an operation that performs a read on your product catalogue and an operation that performs an update may be developed as individual microservices to

cater to different scalability needs. But, the consumers of these operations would like to see them both in a single API since they perform operations on a single entity (products). This makes it necessary to have a single API consisting of two resources (functions) that expose both microservices.

SECURITY

Each of the microservices above may require different levels of authentication and authorization such as OAuth2.0, certificates, mTLS (mutual transport level security), etc. These may apply on communication channels between consumer applications and microservices or on inter-microservice communication channels as well. This makes it harder for applications to consume your microservices as well as making it harder for programming inter-microservice communications. This problem can be solved by using an API Gateway capable of exposing a uniform security mechanism to its consumers and translating them to whatever the necessary internal security protocols are. For example, exposing OAuth 2.0-based security for consumer applications while using mTLS to communicate with internal microservices.

THE MICROGATEWAY PATTERN

As discussed in the section above, the use of an API gateway in an API Management solution helps us to solve most problems related to exposing our microservices to consumer applications. One key factor to keep in mind here is that the main reason an organization adopts microservices is because of the agility they provide in terms of development, testing, deployment, and scaling applications. As businesses keep innovating, they will end up with more microservices and more integrations between them. This will only result in us seeing more APIs, which raises a question on our API Gateway architecture. Just like microservices, this situation demands our API gateways be agile in terms of development, testing, deployment, and scalability. Instead of hosting the runtime of all our APIs on a single monolith gateway, this situation demands the capability of deploying each individual API or smaller subsets of APIs on individual Microgateways.

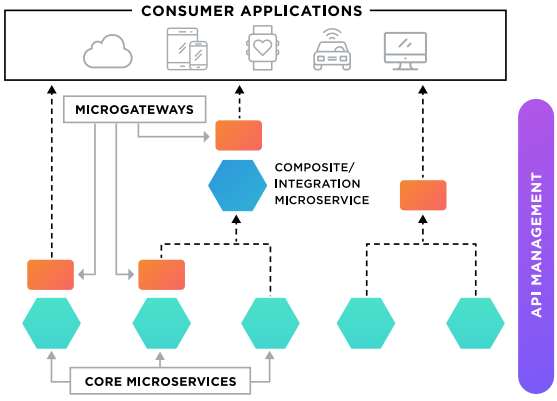


Figure 2: A microservices Architecture Influenced by Microgateways

Some key characteristics of Microgateways in order to meet this criteria are described below.

DESIGNED TO SCALE

The ability to scale easily to meet rising demand is a critically important factor for a Microgateway. As such, it is important that the Microgateway does not connect to any other systems at runtime so that it can scale without resistance.

CI/CD READINESS

The creation, deployment, and spinning down of Microgateways should be as trivial as possible. These processes should be automated in their entirety, meaning that they should entertain CLI-like tools or administrative APIs so that these processes can be performed by automation tools such as Jenkins during their build or deployment phases.

CONTAINER READINESS

Many of today's software following microservices standards are being deployed on containers and container orchestration systems for the range of benefits they offer. As such, the gateways that become the entry point into the services offered by this software should bear characteristics that make them easy to deploy on Containers. Some of these are:

- Small distribution size (<100 MB)
- Low resource consumption and fast boot-up (<256 MB, 1 second boot up)
- Configurable via environment variables
- Build processes that generate container images and deployment artifacts, and the ability to automate them

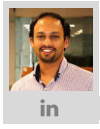
IMMUTABILITY

Microservices architectures demand scalability and automated recovery. This makes the predictability of software very important so that we guarantee that we deploy and scale the exact artifact we have tested. A given Microgateway runtime needs to be built once with its APIs and policies so that we guarantee its replicas are identical.

CONCLUSION

As organizations steadily embrace microservices and as businesses keep on innovating, we are looking at a future where the number of APIs being used within and exposed out of an organization are due for a rapid increase. In such a landscape, it is important for us to have scalable mechanisms for the development, deployment, and management of these APIs. The management of these APIs are going to be a lot more than management of API proxies. Being able to compose integrated/composite microservices, expose them as Microgateways, and manage them via API Management systems efficiently are going to play a critical role on how agile and competitive an organization can be. Gearing up for this growth should definitely be in your plans!

NUWAN DIAS works as a Director at WSO2, an open source integration provider. He is a member of the architecture team at WSO2, which spearheads the strategy and design of WSO2 products. He spends most of his time working closely with the engineering team, which is responsible for the research and development of the WSO2 API Manager. Nuwan is a member of the Open API Security Group and has spoken at numerous conferences around the world on the topics of APIs, integration, security, and microservices.




WSO2
API Manager
An open source approach to addressing any spectrum of the API lifecycle, monetization and policy enforcement.
wso2.com/api-management

The Rise of the Composable Enterprise

Current IT trends show that over the next few years, enterprises will find they need to deal with more than 1 trillion programmable endpoints and APIs. These will consist of traditional application APIs, data APIs, data streams, software component APIs, microservices, sensors, and IoT inputs. Indeed, **everything may become an API**.

Knowledge workers know this, and will need access to all these APIs and endpoints whether it's only to create a basic SaaS-to-SaaS connection or to create a more complex integration. Therefore, over the next 2 to 5 years, we expect that tools and processes will necessarily evolve to address this level of scale and complexity.

Additionally, infrastructures to support this huge quantity of endpoints will gravitate toward those optimized for microservices and serverless underpinnings. From a development perspective, current low-code integration approaches that involve centralized IT orgs or waterfall style processes simply will not scale. As a result, architectures will tend toward more decentralized, cell-based approaches underpinned by microservices and serverless solutions.

With the trend toward trillions of endpoints, APIs will serve as the core of digital apps and applications that rely on IoT data and artificial intelligence (AI). This is at the core of the disruption WSO2 sees in the coming years: that IT organizations tend less toward "development," and more toward being "API integrators." We call this new disruptive IT phase the **composable enterprise**, which will be fueled by the explosive availability and use of APIs and programmable endpoints.

For more industry observations and implications, read our blog.

RELATED RESOURCES:

- Introduction to WSO2 API Microgateway
- API Microgateway walkthrough

WRITTEN BY KEN OESTREICH
VICE PRESIDENT - PRODUCT MARKETING, WSO2

PARTNER SPOTLIGHT

WSO2 API Manager

An open source approach to addressing any spectrum of the API lifecycle, monetization, and policy enforcement.

CATEGORY
API Management and Microservices

RELEASE SCHEDULE
Quarterly

OPEN SOURCE?
Yes

CASE STUDY

to provide their services to customers in the digital experience of their choice. Rather than waiting for the customer to come to them, they wanted to take financial services to customers where they most needed them. By taking on this initiative, they became the first large bank in the USA to build an API platform, which won them the overall most innovative award at the 2018 Monarch Innovation Awards.

The Wells Fargo API Gateway, powered by WSO2 API Manager, has over 20 APIs in production for various data services and payment functions such as account aggregation, account balance, foreign exchange, and wire payments. With their developer portal, partners can easily integrate with the APIs, sometimes even in just a day. The portal is very developer-friendly with an abundance of resources including code snippets, Swagger documentation, and easy access to keys and application registration.

Since its release in 2016, API adoption is off the charts, making them the first (of large banks) and biggest provider of APIs through an API channel in the financial sector. With the help of WSO2 products and services and by partnering with third-party providers, Wells Fargo was able to embed financial services into the daily lives of everyday people.

Listen to their story here.

STRENGTHS

1. Open source
2. Broad API management solution that offers cloud, on premise, or hybrid
3. Designed to fit into monolithic and microservice architectures with gateway and microgateway capabilities

NOTABLE USERS

- Wells Fargo
- Stubhub
- Hospital Corporation of America
- Brigham Young University
- CitySprint
- Proximus

WEBSITE wso2.com **TWITTER** @wso2 **BLOG** wso2.com/blogs/thesource

Designing a Usable, Flexible, Long-Lasting API

QUICK VIEW

01. The most commonly missed steps for API design are the basics: start with who, what, where, why, and how.

02. Take advantage of best practices including properly using nouns, HTTP methods, status codes, and descriptive error messages.

03. Hypermedia is critical to remove business logic from the client, allowing for server-client separation and flexibility.

BY MIKE STOWE
HEAD OF DEVELOPER MARKETING, **RINGCENTRAL**

I remember the final commit like it was yesterday, even though it was several years ago. Six months of work building our application, and we were ready to launch. It followed all of the best practices, the code was perfect (OK, maybe adequate), and it was just in time for the big release. Like an episode of Silicon Valley, we waited to for the numbers — numbers that never came. Instead, after six months of work, we watched as customers expressed interest and then just walked away.

We designed a solution our customers didn't need. Panic set in, and we spent the next three to four weeks quickly refactoring the code, trying to make it something they could use. But it was too little, too late. In the end, the company spent around $1 million on a project that was discarded in the trash, all because we forgot one simple rule: **Building an API is easy. Designing a usable, flexible, long-lasting API is hard.**

While this experience sticks with me, it also surprises me how so many small and large businesses alike forget this simple rule. If you don't understand what it is you're building, why you're building it, or who you're building it for — how can it truly meet their needs? How can it meet your needs? How can it be successful?

WHO, WHAT, WHERE, WHY, AND HOW

Before your design your API, you need to know who you're building it for. Is it for internal consumers, customers, third-party developers, or all of the above?

Once you answer the *who*, the next question becomes simpler: *What* do they need access to? Commonly, this step gets confused with the HTTP methods (GET, POST, PUT, PATCH), but instead, at this junction, you should be able to list what actions they need.

As you start thinking about the actions your users need to be able to do, place them in a chart, like so:

Users	Create user, edit user, reset password, suspend user, message user
Messages	Create draft, send message, read message, delete message

This chart, while rudimentary, offers significant advantages from the get-go. If you're building a RESTful API, you've already identified your resources (in this case, users and messages). The chart also forces you to think through the workflows. Finally, it helps reduce duplication by making us think through where certain endpoints should live; in this case, it doesn't make sense to have an endpoint for messaging users under */users* if we have a resource */messages*. It does, however, make sense to have a link from the users object to the message (allowing us to build out a hypermedia map as we go).

The next step of the process is to understand *why* we're building the type of API we are. This step is important — it forces us to consider the specific reasons we're choosing that format over others and to understand the format we're choosing.

SOAP	RPC	REST
• Requires a SOAP library on the end of the client • Not strongly supported by all languages • Exposes operations/ method calls • Larger packets of data, XML format required • All calls sent through POST • Can be stateless or stateful • WSDL - Web Service Definitions • Most difficult for developers to use	• Tightly coupled • Can return back any format, although usually tightly coupled to the RPC type (e.g. JSON-RPC) • Requires user to know procedure names • Specific parameters and order • Requires a separate URL/ resource for each action/ method • Typically utilizes just GET/POST • Rquires extensive documentation • Stateless • Easy for developers to get started	• No library support needed, typically used over HTTP • Returns data without exposing methods • Supports any content-type (XML and JSON used primarily) • Single resource for multiple actions • Typically uses explicit HTTP action verbs (CRUD) • Documentation can be supplemented with hypermedia • Stateless • More difficult for developers to use

Most APIs aren't truly REST APIs, so if you choose to build a RESTful API, do you understand the constraints of REST including hypermedia/HATEOAS? If you choose to build a partial REST or REST-like API, do you know why you are choosing to not follow certain constraints of REST?

Depending on what your API needs to be able to do and *where* your API will be used, legacy formats such as SOAP may make sense. However, with each format comes a tradeoff in terms of usability, flexibility, and development costs.

Finally, as we start to plan our API, it's important to understand *how* our users will interact with the API and *how* they'll use it in conjunction with other services. Be sure to use tools like RAML or Swagger/OAI during this process to involve your users, provide mock APIs for them to interact with, and to ensure your design is consistent and meets their needs.

BEST PRACTICES

As you design your API, it's also important to remember that you're laying a foundation to build upon at a later time. One of the easiest ways to kill an API is to try to reinvent the wheel or improve upon existing standards/ideas in production without heavily testing them first.

In other words, as a developer, don't get fancy. Build on tested and tried best practices that your consumers will be expecting within your API.

NOUNS VS. VERBS

When building a RESTful or REST-like API, utilize nouns as resources. You should be using */users* and then relying on the HTTP methods instead of relying on */getUser* or */deleteUser*.

Another, more controversial best practice is to use plural naming conventions for any resources that return collections. Keep in mind that even if the resource only currently returns a single item, if there's the possibility to return multiple items, then it should be created as a collection.

For example, you might have a resource */address* that only returns one address. But what happens when you allow for a billing address, shipping address, or multiple addresses? From day one, you should call the resource */addresses* and return the address back in an array so that the application evolves to eventually allow for multiple addresses and the contract remains intact.

ACCEPT AND CONTENT-TYPE HEADERS

A best practice to avoid costly refactors is utilizing accept and content-type headers. Regardless of whether your API only accepts JSON or XML today, there may come a time when you're required to add additional formats.

By building in the context-switcher from the start, adding any new formats is as simple as adding the appropriate libraries behind-the-scenes and allowing that format type, letting you easily support multiple formats without refactoring your RESTful API.

USE OF HTTP METHODS

Because actions of a RESTful or REST-like API are dictated by HTTP methods, it's important to understand what each method does and when to use it. Perhaps it's easiest to use CRUD:

- CREATE - POST
- READ - GET
- UPDATE - PUT, PATCH
- DELETE - DELETE

While many of the HTTP methods seem pretty clear, the RFC standards allow for unique usage for each. For example, POST is a magical method that *can* be used for most operations; however, that doesn't mean that it *should*. Instead, rely on POST to create a new item in a collection or create a new record. Also, while POST makes sense on a collection, it may not make sense at an item level.

PUT is another unique method — it can be used to create a new record if it doesn't exist (but only if an explicit ID is provided, the record doesn't exist, and a 201 status is returned), or used to update an existing record. It's important to remember that many developers aren't familiar with the create aspect of PUT, and many frameworks don't test to see if the result is a 200 or 201. As such, they may receive a successful response when expecting a 404 Not Found. Be very wary of using PUT to create new records, even in those specific situations, unless necessary and well-documented within your API.

PATCH is widely suggested as an alternative to PUT, as it will PATCH the data record with the data it receives.

Also keep in mind that you probably do not want to allow PUT, PATCH, or DELETE on a collection, as that would essentially update or delete all of the items in that collection.

DESCRIPTIVE ERROR MESSAGES

Along with understanding how to use your API, it's important to understand why something didn't work when using your API. I can't tell you the number of APIs I've used where the error message is simply "something went wrong" or "invalid parameter." These messages do nothing to assist the developer in using your API.

Instead, provide clear, relevant error messaging. Don't just tell the user that an ID is missing — tell them *why* they need the ID to begin with. If there's an internal support code that they can use when emailing your team, give them that code. Most importantly, provide a link to documentation that shows them how to fix the

error. You don't have to recreate the wheel or guess what should be in the error message.

There are several great formats that already exist today including Google Errors, JSON API, and vnd.error.

USE HYPERMEDIA

While HATEOAS (Hypermedia as the Engine of Application State) sounds extreme, its purpose is simple: remove the business logic from the client.

Think of how APIs are built today: The client receives a response, and based on that response, needs to determine what actions it can take against the user. It can do this by utilizing the OPTIONS method (if the API allows for it) or making the calls to see if they work (getting a 200 or 400), or the developer can mimic the business logic on the client and hope nothing changes.

HATEOAS uses links to tell the client what actions can be taken next. If a business rule changes, the link changes — meaning there's no rework on the client or developer's end. This is crucial because it also allows for the separate evolution of the client and server.

Take, for example, three users: one is an admin, one is a standard user, and one was caught spamming. Most likely, the application state of each object is different. The super user can't be deleted, the standard user can be suspended, and the suspended user can't be suspended again. The links returned in the response explain the state of each one of these users in a way that doesn't require the client to know the underlying business rules.

Perhaps the most common example of hypermedia can be found on the world wide web. If you visit DZone.com, you don't have to type in the URL of the article you're looking for. You go to DZone.com, click a link, click another link, and eventually end up where you want to be. This all occurs because of HTML (hypertext markup language).

While HTML is perhaps the most common form of hypermedia, there are several popular formats you can use for your API today, including HAL and JSON API. As these specifications mature, we're also seeing newer formats like Siren and CPHL — which extend the HAL format with the HTTP methods, forms, code on demand, and flexible documentation.

Example of HAL/CPHL in an API:

```
{
    "firstName": "Mike",
    "lastName": "Stowe",
    "_links": {
        "edit": {
            "href": "/users/1",
            "methods": ["put", "patch"]
        },
        "message": {
            "href": "/message?to=1",
            "methods": ["post"]
        }
    }
}
```

Hypermedia also allows for contractual flexibility. Say you have a messaging resource and hypothetically, one of your users accidentally places it in an infinite loop, causing 10,000 sends before hitting their limit. To solve this issue, you add a token to the resource — but if the resource is hard-coded, you've broken the contract, meaning you need to version your API and all of your clients need to update their code.

With hypermedia, since they're relying on a key:value, the URL can change or the token can be added without breaking the contract or causing any inconveniences for your users.

In other words, it's a simple change to the above response like so that lets you put in an additional check without breaking your API's contract:

```
"message": {
    "href": "/
message?to=1&token=893hdy83ikaherukaehfiwf7w48ika",
    "methods": ["post"]
}
```

TO RECAP

Building an API is easy but designing an API that is flexible enough to last years, evolving as your platform evolves, is hard — let alone adding in usability. These best practices, along with tools like RAML or Swagger/OAI can help you make that challenge a reality. But it's up to you to ensure that you take the time, involve your users, and think through every aspect of your API. Like a foundation, one stone in the wrong place can cause the whole thing to tumble — and with an API, it only takes *one* little thing to significantly reduce its lifespan.

MIKE STOWE Author of Undisturbed REST and an international speaker, Mike Stowe is a leading proponent of API design and emerging technologies. Serving as the Head of Developer Marketing for RingCentral, his mission is to not only create better APIs, but improve the overall developer experience. You can find more articles, talks, and his slides at mikestowe.com.

CODE BLOCK CONTINUED ON FOLLOWING COLUMN

Executive Insights on the Current and Future State of API Management

QUICK VIEW

01. The most important element of API management is their security — including availability and access.

02. APIs have made the creation of applications faster, resulted in more flexible applications, and given the developers the opportunity to reuse code.

03. The most common issue of API management is lack of attention on security or thinking APIs can be secured like applications.

BY TOM SMITH
RESEARCH ANALYST AT **DZONE**

To gather insights on the current and future state of API management, we talked to 17 executives who are using APIs in their own organization, as well as helping clients use APIs to accelerate their digital transformation and the development of quality applications. Specifically, we talked to:

- Maxime Prades, Vice President of Product, Algoria
- Jaime Ryan, Senior Director, Product Management & Strategy API Management, CA Technologies
- Ross Garrett, VP Marketing, Cloud Elements
- OJ Ngo, CTO, DH2i
- Reid Tatoris, Vice President Product Outreach and Marketing, Distil Networks
- Oren Novotny, Chief Architect, DevOps and Modern Software, Digital Innovation, Insight
- Raj Sabhlok, CEO, ManageEngine
- Keith Casey, API Problem Solver, Okta
- Vikas Anand, Vice President Product Development, Oracle
- Mike LaFleur, Global Director Solution Architecture, Provenir
- Steve Willmott, Senior Director and Head of API Infrastructure, Red Hat
- Keshav Vasudevan, Product Marketing Manager, SmartBear
- Chris McFadden, V.P. of Operations, SparkPost
- Jerome Louvel, VP of Product Management, Talend
- Derek Birdsong, Product Marketing Manager, Connected Intelligence Cloud, TIBCO
- Setu Kulkarni, Vice-President of Product and Corporate Strategy, WhiteHat Security
- Roman Shaposhnik, Co-founder VP Product Strategy, Zededa
- Vijay Tapaskar, Co-founder VP Engineering and Ops, Zededa

KEY FINDINGS

1. The most important element of API management is **security — including availability and access**. It's important to understand that an API can be abused just like an end-point, website, or application, but it requires different security protocols including controlling access through analytic security policies. API security should be the first feature of an API management product.

Documentation and standards are also needed around the creation of the APIs and the business logic so there is no need for an external repository or documents management center and so teams creating multiple APIs for the same organization can provide the same user experience (UX) across APIs.

2. APIs have made the creation of applications **faster, resulting**

in more flexible applications, and have given developers the opportunity to reuse code. It's fundamentally easier and faster to create composite applications that pull in data capabilities from a wide range of sources. APIs allow developers to work across different platforms and enable the use of the microservices pattern from monolithic applications to individual elements using APIs to call between and enable microservices. Most APIs are being used by tens to hundreds of developers internally and thousands to millions externally.

3. OAuth is the most popular industry protocol for securing APIs. OAuth is the primary method of access control for delegated access to APIs. For instance, when you allow an app to log in using your Facebook identity, or get access to your photos on Instagram, there is a carefully controlled three-way handshake that allows you to grant permission to that app for a specific scope of access. A layered approach is recommended by several respondents along with good coding practices and standards, automated testing, pen testing, patch management, and regular audits through SOC2 and internal teams.

4. Real-world problems being solved by APIs run across multiple industries with **airlines and airports being mentioned most frequently** followed by financial services. Airlines are using API platforms to enable third parties to improve end-to-end customer travel experiences and focus on operational improvements, like flight operations automation and leveraging back office data more effectively in airport operations. The Amsterdam airport is using APIs to monitor applications to improve customer experience (CX).

5. The most common issues with API management are the **lack of attention on security and thinking that APIs can be secured like applications**. You cannot give access to just anyone, you must tighten access and authentication to the individual. Follow the security standards set by Google for APIs — SSO and REST. Put authentication in place with OAuth or Open ID.

6. Concerns around the current state of API management revolve around **security and the consolidation of third-party tools**. There is a lack of focus on security. A lot of customers use WAF or CDN, but these are not able to stop automated attacks. APIs are open to all kinds of malware, so every API needs to be certified as secure.

There is a lot of consolidation of tools and this will come at the expense of end-user experience. We will see movement from the core value of the tool to generate more revenue without regard to building and testing. This will hurt innovation as we've seen with databases.

7. The future of API management is around **security as it relates to access and standards**. More standardization will weed out those who are not following standards and principles. Security and privacy will be audited more thoroughly, and we will have independent security standards for APIs. There will be a greater focus on identity and limited access to APIs by understanding the use cases the APIs are being designed for.

Team collaboration is very important. There is a need for API governance that begins with the design itself and an API blueprint for developers working on APIs and delivering APIs that provide consistent rules of engagement. API platforms will help ensure consistency and compatibility.

APIs are becoming more focused on delivering a particular solution. Understand your use case and build the most elegant solution.

8. When managing APIs, developers need to think about **security, purpose, and the end-user (i.e. other developers)**. Be aware of the security implications of what you are building. Think like a distributed systems' engineer. A key does not equal security, because credentials can be stolen. APIs are not under attack any less than your site or applications; however, they do need a different type of security. Make sure you have continuous security monitoring. API security is actually more important than general web security.

Are you delivering an API that's fit for purpose? APIs are becoming more focused on delivering a particular solution. Understand your use case and build the most elegant solution. Know what you want the CX to be and focus on the technology solutions you want to provide. Think about the purpose of the API you are building, who will access it, and whether they are internal or external. People will use the APIs that give them access to the data or systems they want. Your API will be used for things you cannot predict — be prepared.

TOM SMITH is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

Solutions Directory

This directory of API management platforms, ESBs, integration platforms, and frameworks provides comprehensive, factual comparisons of data gathered from third-party sources and the tool creators' organizations. Solutions in the directory are selected based on several impartial criteria, including solution maturity, relevance, and data availability.

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
1060 Research	NetKernel	Integration platform	N/A	1060research.com
Actian	DataCloud	Integration platform/PaaS	Demo available by request	actian.com
Adaptris	Interlok	Integration platform/PaaS	N/A	adaptris.com/interlok.php
Adeptia	Adeptia Integration Suite	Integration platform/PaaS	30 days	adeptia.com/products/adeptia-integration-suite
AdroitLogic	UltraESB-X	ESB	Open source	adroitlogic.org/products/ultraesb
Amazon	Amazon SQS	Message queue	Free tier available	aws.amazon.com/sqs
Apache Software Foundation	ActiveMQ	Message queue	Open source	activemq.apache.org
Apache Software Foundation	Camel	Integration framework	Open source	github.com/apache/camel/blob/master/README.md
Apache Software Foundation	Qpid	Message queue	Open source	qpid.apache.org
Apache Software Foundation	ServiceMix	Integration container/platform	Open source	servicemix.apache.org
Apache Software Foundation	Synapse	ESB	Open source9	synapse.apache.org

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Apigee	Apigee Edge	API management	Free tier available	docs.apigee.com/api-platform/get-started/what-apigee-edge
Aurea Software	CX Messenger	ESB	Free tier available	aurea.com/our-solutions/experience-solutions/cx-messenger
Axway	Axway API Management	API management	N/A	axway.com/en/enterprise-solutions/api-management-solutions
Azuqua	Azuqua Platform	Integration platform/PaaS	30 days	azuqua.com
BroadPeak	K3	Integration platform/PaaS	Demo available by request	broadpeakpartners.com/k3/features
Built.io	Built.io Flow	Integration platform/PaaS	30 days	built.io/products/flow
CA Technologies	CA API Management	API management	Available by request	ca.com/us/products/api-management.html
Cazoomi	Cazoomi	API management	Free tier available	cazoomi.com
Cleo	Cleo Integration Cloud	Integration platform	Demo available by request	cleo.com/cleo-integration-cloud
Cloud Elements	API Manager Platform	API management	Available by request	cloud-elements.com/platform/
Cloud Native Computing Foundation	NATS	Message queue	Open source	nats.io
Dell Boomi	AtomSphere	Integration platform/PaaS	Free tier available	boomi.com/integration/integration-technology
DreamFactory Software, Inc.	DreamFactory	API management	30 days	dreamfactory.com
Enduro/X	Enduro/X	Integration platform	Open source	endurox.org
Fanout, Inc.	Fanout Zurl	Integration platform	Open source	github.com/fanout/zurl
Fiorano Software	Fiorano API Management	API management	Demo available by request	fiorano.com/products/fiorano_api

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Fiorano Software	Fiorano ESB	ESB	Free solution	fiorano.com/products/fiorano_esb
Fiorano Software	Fiorano Integration	Integration platform	Free solution	fiorano.com/products/fiorano_integration
Fiorano Software	FioranoMQ	Message queue	Free solution	fiorano.com/products/fiorano_mq
Flowgear	Flowgear	Integration platform/PaaS	Available by request	flowgear.net
Fujitsu	RunMyProcess	Integration platform/PaaS	Demo available by request	runmyprocess.com/platform
IBM	IBM API Management	API management	Free tier available	ibm.com/software/products/en/api-connect
IBM	IBM Integration Bus	ESB	30 days	ibm.com/support/knowledgecenter/en/SSMKHH_9.0.0/com.ibm.etools.mft.doc
IBM	IBM MQ	Message queue	Free developer tier	ibm.com/products/mq
iMatix Corporation	iMatix Suite	Message queue	Open source	github.com/imatix
Information Builders	Data Management Platform	Integration platform	Available by request	informationbuilders.com/products/data-management-platform
Instant API	Instant API	API creation & hosting	N/A	instantapi.com
integrator.io	Celigo	Integration platform/PaaS	Free tier available	celigo.com/ipaas-integration-platform
InterSystems Corporation	Ensemble	ESB	N/A	docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page
Iron.io	IronMQ	Message queue	Available by request	iron.io/mq
Jitterbit	Jitterbit	Integration platform/PaaS	Available by request	jitterbit.com

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
JNBridge, LLC	JNBridge Adapters	Integration platform	30 days	jnbridge.com
Liaison Technologies	Liaison Alloy	Integration platform/PaaS	Demo available by request	liaison.com/liaison-alloy-platform
Mertech Data Systems, Inc.	evolution	API management	Free tier available	thriftly.io
Micro Focus	Artix	ESB	30 days	microfocus.com/products/corba/artix#
Microsoft	BizTalk Server	Integration platform/PaaS	180 days	microsoft.com/en-us/cloud-platform/biztalk
Microsoft	Microsoft Azure API Management	API management	Free tier available	azure.microsoft.com/en-us/services/api-management
MID GmbH	Innovator Enterprise Modeling Suite	SOA governance	60 days	mid.de/en/business-activities/tools/innovator
Moltin	Moltin	eCommerce API	30 days	moltin.com
MuleSoft	Mule ESB	ESB	Available by request	mulesoft.com/platform/soa/mule-esb-open-source-esb
MuleSoft	Anypoint Platform	API management	Available by request	mulesoft.com/platform/enterprise-integration
Neuron ESB	Neuron ESB CU4	ESB	30 days	neuronesb.com
OpenESB	OpenESB	ESB	Open source	open-esb.net
Oracle	Oracle Service Bus	ESB	Free solution	oracle.com/technetwork/middleware/service-bus/overview
Oracle	Oracle SOA Suite	SOA governance	30 days	oracle.com/middleware/application-integration/products/soa-suite.html
OW2 Middleware Consortium	Petals ESB	ESB	Open source	petals.linagora.com
Particular Software	NServiceBus	ESB	Open source	particular.net/nservicebus

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Pivotal Software, Inc.	RabbitMQ	Message queue	Open source	network.pivotal.io/products/pivotal-rabbitmq
Pivotal Software, Inc.	Spring Integration	Integration framework	Open source	spring.io/projects/spring-integration
PokitDot, Inc.	Pokitdot	Healthcare API management	30 days	pokitdok.com
Red Hat	3Scale API Management	API management	90 days	3scale.net/api-management
Red Hat	Fabric8	Integration platform/PaaS	Open source	fabric8.io
Red Hat	JBoss Fuse	ESB	Open source	redhat.com/en/technologies/jboss-middleware/fuse
Redis Labs	Redis	Database and message queue	Open source	redis.io
RoboMQ	RoboMQ	Integration platform/PaaS	90 days	robomq.io
Rocket Software, Inc.	Rocket Data	Data integration platform/PaaS	N/A	rocketsoftware.com/products/rocket-data
Rocket Software, Inc.	LegaSuite Integration	Integration platform/PaaS	Demo available by request	rocketsoftware.com/products/rocket-legasuite/rocket-api
Rogue Wave Software	Akana API Management	API management	Demo available by request	roguewave.com/products/akana/solutions/api-management
Rogue Wave Software	Akana Software Suite	SOA governance	Demo available by request	roguewave.com/products-services/akana/solutions/integrated-soa-governance
Runscope	Runscope	API monitoring	Available by request	runscope.com
SAP	SAP API Business Hub	API management	N/A	api.sap.com
Scheer	E2E Bridge	Integration platform/PaaS	30 days	e2ebridge.com/en
SEEBURGER	Seeburger	Integration platform/PaaS	N/A	seeburger.eu/business-integration-suite.html
Sikka Software Corporation	Sikka One API	API management	Demo available by request	sikkasoft.com

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
SmartBear Software	SoapUI NG	API testing	14 days	smartbear.com/product/ready-api/soapui-ng/overview
SmartBear Software	LoadUI	API load testing	14 days	smartbear.com/product/ready-api/loadui/overview
SmartBear Software	Swagger	Integration framework	Free tier available	swagger.io
SnapLogic, Inc.	Elastic Integration Platform	Integration platform/PaaS	Demo available by request	snaplogic.com/enterprise-integration-cloud
Software AG	WebMethods	Integration suite	Available by request	softwareag.com/corporate/products/az/webmethods
StormMQ	StormMQ	Message queue	Open source	github.com/stormmq
SwarmESB	SwarmESB	ESB	Open source	github.com/salboaie/SwarmESB
Talend	Restlet Platform	API management	Free tier available	restlet.com
Talend	Talend ESB	ESB	Available by request	talend.com/products/application-integration
Talend	Talend Integration Cloud	Integration platform/PaaS	30 days	talend.com/products/integration-cloud
TIBCO Software Inc.	TIBCO Cloud Integration	Integration platform/PaaS, API management	Available by request	tibco.com/products/tibco-cloud-integration
TIBCO Software Inc.	Mashery	API management	30 days	mashery.com/api-management/saas
Tyk.io	Tyk	API management	Open source	github.com/TykTechnologies/tyk
Vigience	Vigience Overcast	Salesforce integration platform	Available by request	overcast-suite.com
WSO2	WSO2 Carbon	Integration platform/PaaS	Open source	wso2.com/products/carbon
WSO2	WSO2 Enterprise Integrator	ESB	Open source	wso2.com/products/enterprise-service-bus
WSO2	WSO2 Message Brokers	Message queue	Open source	wso2.com/products/message-broker
Zapier	Zapier	API management	Free tier available	zapier.com

GLOSSARY

API
A software interface that allows users to interact with & configure other programs, usually by calling from a list of functions.

DENIAL OF SERVICE (DOS) ATTACK
An attack on a resource, caused when a perpetrator intends to make a resource unavailable by placing massive amounts of requests on the given resource.

DOCUMENTATION-DRIVEN DEVELOPMENT
A philosophy of software development in which documentation for a feature is written before the feature is created.

ENTERPRISE INTEGRATION (EI)
A field that focuses on interoperable communication between systems and services in an enterprise architecture; it includes topics such as electronic data interchange, integration patterns, web services, governance, & distributed computing.

ENTERPRISE INTEGRATION PATTERNS (EIP)
A growing series of reusable architectural designs for software integration; frameworks such as Apache Camel & Spring Integration are designed around these patterns, which are largely outlined on EnterpriseIntegrationPatterns.com.

ENTERPRISE SERVICE BUS (ESB)
A utility that combines a messaging system with middleware to provide comprehensive communication services for software applications.

FARMING
The idea of providing services by leveraging the services of another resource or resources.

GRAPHQL
A query language & runtime for completing API queries with existing data.

HYPERTEXT TRANSFER PROTOCOL (HTTP)
A protocol used to exchange hypertext; the foundation of communication for websites & web-based applications.

INTEGRATION FRAMEWORK
A lightweight utility that provides libraries & standardized methods to coordinate messaging among different software.

IPAAS
A set of cloud-based software tools that govern the interactions between cloud & on-premises applications, processes, services, & data.

MESSAGE BROKER
Middleware that translates a message sent by one piece of software to be read by another piece of software.

MICROSERVICES
Small, lightweight services that each perform a single function according to a domain's bounded contexts; the services are independently deployable & loosely coupled.

MIDDLEWARE
A software layer between the application & operating system that provides uniform, high-level interfaces to manage services between distributed systems; this includes integration middleware, which refers to middleware used specifically for integration.

REPRESENTATION STATE TRANSFER (REST)
A set of principles describing distributed, stateless architectures that use web protocols & client/server interactions built around the transfer of resources.

RESTFUL API
An API that is said to meet the principles of REST.

SERVICE DISCOVERY
The act of finding the network location of a service instance for further use.

SERVERLESS COMPUTING
A cloud computing model in which a provider manages the allocation of servers & resources.

SERVICE-ORIENTED ARCHITECTURE (SOA)
An application architecture built around the use of services that perform small functions.

SIMPLE OBJECT ACCESS PROTOCOL (SOAP)
A protocol that is used by web services to communicate with each other, commonly used with HTTP.

STATELESS SESSION STATE
A session which no information is maintained between the sender & receiver.

SWAGGER
A definition format used to describe & document RESTful APIs in order to create a RESTful interface to develop & consume APIs.

WEB SERVICE
A function that can be accessed over the web in a standardized way using APIs that are accessed via HTTP & executed on a remote system.

INTRODUCING THE

Open Source Zone

Start Contributing to OSS Communities and Discover Practical Use Cases for Open-Source Software

Whether you are transitioning from a closed to an open community or building an OSS project from the ground up, this Zone will help you solve real-world problems with open-source software.

Learn how to make your first OSS contribution, discover best practices for maintaining and securing your community, and explore the nitty-gritty licensing and legal aspects of OSS projects.

COMMITTERS & MAINTAINERS

COMMUNITY GUIDELINES

LICENSES & GOVERNANCE

TECHNICAL DEBT

Visit the Zone

BROUGHT TO YOU IN PARTNERSHIP WITH Flexera