THE DZONE GUIDE TO

INTEGRATION

MICROSERVICES, APIs, AND PLATFORMS

VOLUME III

BROUGHT TO YOU IN PARTNERSHIP WITH

IBM

3scale BY RED HAT

Dear Reader,

50 years ago, Star Trek predicted a world where people had such access to information that they could be whatever they wanted and change the world for the better. In this year's *Guide to Integration: Microservices, APIs, and Platforms*, we look at how the world of integration brings the availability of data to new levels and helps make it possible for anyone, anywhere to get access to information they never had before. In short – APIs are the mortar that help combine the services we use on a daily basis.

Integration services and APIs are specifically powering the growth of Mobile and IoT and are helping people and organizations solve problems that would have been impossible just a few short years ago. Organizations are pushing the envelope to make our world a better place. Medic Mobile, a nonprofit organization, uses mobile and APIs to help medical workers in Africa bring healthcare to remote villages, and Vinli, a company that uses APIs to turn all cars into smart cars are just two examples.

APIs also help us accelerate the pace at which we can access information, allow us to communicate with other people around the globe, and help keep information free. Through the login APIs provided by Facebook and Twitter, we have one less login and password to remember, and through those same services we can communicate and share information with people from around the globe. APIs allow us to more easily transfer our information between vendors, which provides a survival of the fittest process across vendors and allows us to use the best tool at any given time.

At an ever increasing rate, people have access to data from a wide variety of services, both public and private, and can use that data to create new services and provide new views on those services. In this guide, we focus on the enterprise's access to data and the use of APIs to better serve their customers. As a specific example, here at DZone we're big fans of using APIs. In fact, we leverage dozens every day along with first person access to data to give everyone visibility into the trends and KPIs of all of our different teams. Through this availability of information, we are able to leverage everyone in the company to make better decisions and help suggest ways to move us forward.

Thank you loyal DZone reader for taking the time to read this guide, and thank you to all of our members who contributed to the guide, helped create the survey, and reviewed draft versions of it. Without you, we wouldn't be where we are today. Now, on behalf of everyone here at DZone, please enjoy the *2017 Guide to Integration: Microservices, APIs, and Platforms*.

BY MATT SCHMIDT
PRESIDENT AND CTO, DZONE
RESEARCH@DZONE.COM

TABLE OF CONTENTS

EDITORIAL

CAITLIN CANDELMO
DIRECTOR OF CONTENT + COMMUNITY

MATT WERNER
CONTENT + COMMUNITY MANAGER

MICHAEL THARRINGTON
CONTENT + COMMUNITY MANAGER

NICOLE WOLFE
CONTENT COORDINATOR

MIKE GATES
CONTENT COORDINATOR

SARAH DAVIS
CONTENT COORDINATOR

INDUSTRY + VENDOR RELATIONS

TOM SMITH
RESEARCH ANALYST

BUSINESS

RICK ROSS
CEO

MATT SCHMIDT
PRESIDENT & CTO

JESSE DAVIS
EVP & COO

KELLET ATKINSON
DIRECTOR OF MARKETING

MATT O'BRIAN
SALES@DZONE.COM
DIRECTOR OF BUSINESS DEVELOPMENT

ALEX CRAFTS
DIRECTOR OF MAJOR ACCOUNTS

JIM HOWARD
SR ACCOUNT EXECUTIVE

CHRIS BRUMFIELD
ACCOUNT MANAGER

PRODUCTION

CHRIS SMITH
DIRECTOR OF PRODUCTION

ANDRE POWELL
SENIOR PRODUCTION COORDINATOR

G. RYAN SPAIN
PRODUCTION PUBLICATIONS EDITOR

ART

ASHLEY SLATE
DESIGN DIRECTOR

SPECIAL THANKS to our topic experts, Zone Leaders, trusted DZone Most Valuable Bloggers, and dedicated users for all their help and feedback in making this report a great success.

WANT YOUR SOLUTION TO BE FEATURED IN COMING GUIDES?
Please contact research@dzone.com for submission information.

LIKE TO CONTRIBUTE CONTENT TO COMING GUIDES?
Please contact research@dzone.com for consideration.

INTERESTED IN BECOMING A DZONE RESEARCH PARTNER?
Please contact sales@dzone.com for information.

Executive Summary

Integrating applications together has always been critical for businesses and consumers. Whether you're integrating your performance management tool with your business intelligence solution, or you're letting your users use Facebook to log on to your service, APIs and integration are key to making everyone's lives simpler and more productive. As we've moved through the 2010s, software has changed in even more drastic ways. DevOps is becoming more widespread, encouraging developers to push even thousands of changes to an application per day, containers are starting to be used in production, and the microservices architectures have made applications more modular, scalable, and changeable. As these applications become more fractured and ever-changing, integrating and communicating with every piece becomes more important than ever. So how has the changing landscape affected the modern developer? We surveyed 773 developers to find out how they're adapting these new technologies and how they're using them. In this guide we'll also cover microservices, REST API design guidelines, transactions across REST services, and more.

MICROSERVICES ARE ON THE RISE

DATA Of DZone members who responded to our survey, 67% are using microservices, up from 27% in 2015. 46% of survey respondents are using microservices for specific purposes, while 21% use them without explicitly deciding to.

IMPLICATIONS The adoption of microservices by the developer community has skyrocketed in the past year, though they are still not used everywhere. Microservices have made architectures more modular, which has been very useful as DevOps and application containers have increased in popularity, as changes can be made only to a part of the whole application, so the entire application does not need to be redeployed. In addition, microservices also help solve issues of scalability for cloud-based applications. The number of respondents who either choose to use microservices or who work with them regardless of choice indicates that education on the benefits of microservices in the tech community has had a noticeable impact, and that any perceived "hype" around them has turned into tangible adoption.

RECOMMENDATIONS Microservices offer a solution to application scalability issues, and help to reduce work-in-progress when it comes to making changes to an application. As more and more businesses are moving to microservices, it is imperative for developers to educate themselves on modular architectures, container solutions such as Docker, and container management tools such as Kubernetes or Docker Swarm. For a real-world example of how Spotify is using microservices, read Petter Mahlen's article on page 6.

ADOPTION OF TOOLS HAS SLOWED DOWN

DATA Out of the tools that our survey asked about, only the usage of Swagger and Kafka increased by a statistically significant amount at 9% and 8% respectively. No other tools changed in their usage by more than 5%.

IMPLICATIONS While this could be seen as a stagnation of the industry, it seems more likely that developers are focusing on their chosen technologies, making gradual improvements to their integration needs as opposed to drastic overhauls due to deficiencies in technology. Once a technology is chosen for connecting pieces of applications, it becomes more difficult to pivot away from using that technology without a significant investment in time and building new knowledge.

RECOMMENDATIONS It's crucial that new technologies be explored when possible in order to continuously improve both applications and, eventually, the entire industry, but this may be easier to accomplish for more agile teams or startups who can afford to take more risks. However, for larger companies, continuous improvement may be the better option to take to make sure everything continues to operate efficiently.

JSON BEATS OUT XML ON USER FRIENDLINESS

DATA A huge majority of survey respondents have used both JSON (97%) and XML (93%). Of members who use JSON, 90% said they enjoyed using it, as opposed to XML, where half of those who have used it said they enjoyed it.

IMPLICATIONS First, the difference between those who use XML vs. JSON is not wide enough to make a definite conclusion on which one is used more. However, JSON has definitely won over more developers due to its simplicity and its readability, as it is meant to only be a data format as opposed to a language like XML. As XML is still in widespread use, a large majority of survey respondents still use the language.

RECOMMENDATIONS As JSON is clearly favored by more users than XML, there are opportunities for new technologies to use JSON as the default data serialization format as a way to increase adoption and enjoyment of their tools. In addition, this data means that RESTful web services do not necessarily mean XML over HTTP. However, as XML is still in widespread use and may be for some time, it is still important that developers keep their XML skills sharp.

Key Research Findings

773 software professionals responded to DZone's 2016 Integration survey. Respondent demographics are as follows:

- 70% of respondents identify as developers or development team leads.
- 69% have more 10 years of experience or more as IT professionals. Respondents have an average of 14 years of experience.
- 41% work at companies with headquarters located in Europe; 26% work at companies with headquarters located in the US.
- 19% work at companies with more than 10,000 employees; 32% at companies between 500 and 10,000 employees; and 16% at companies between 100 and 499 employees.

1. SLOW AND STEADY

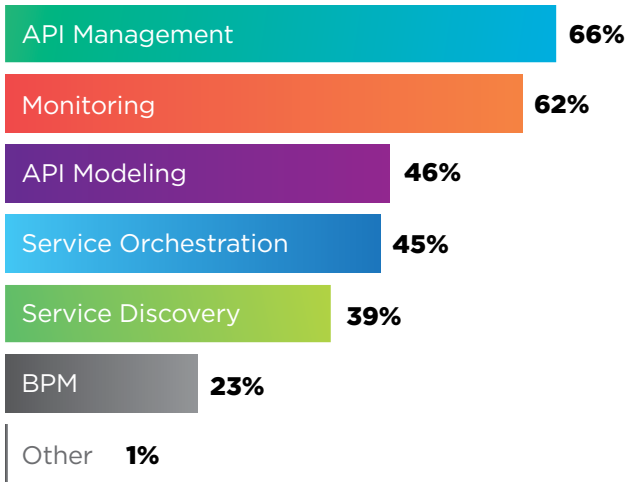
There are a lot of things that haven't changed much in integration over the last year. On average, developers and organizations are generally using the same tools and integrating the same systems. Of the 16 different types of systems we asked our survey respondents about integrating, no system had more than a 5% change from last year's results (ERP integration had the biggest change, increasing 4.9% from last year). Estimated percentages of ad-hoc versus ESB/EAI integration changed only 3%. The use of specific integration topologies, such as point-to-point and hub-and-spoke, changed no more than 2%. And of all the frameworks, iPaaSes, and tools for messaging, API management, and service registry/discovery, only two tools (of sixty total solutions mentioned) had a change of greater than 3%—usage of the Apache Kafka message broker increased 8% from last year, and usage of Swagger for API management and documentation increased 9%. From last year to this year, respondents' answers had many other similarities as well; but this far from signifies a stagnation in software integration. Rather, it seems to signify a determined focus on certain key integration practices and problems, and a slow and steady push to towards improving integration implementation and design.

2. MACRO CHANGE IN MICROSERVICES

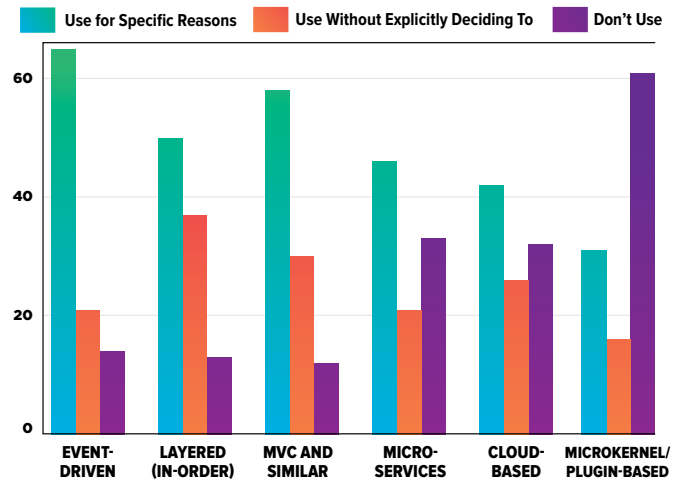
One of the most significant changes we saw in our survey this year was the amount of respondents claiming to use microservices. Last year, 27% of respondents said they or their organization used microservices architectures, down 13% from the year before. This year, two out of three respondents said they use microservices architectures: 46% said they use microservices for specific reasons, and 21% said they use microservices without explicitly deciding to. Part of this drastic change may have to do with a fluid understanding of the definition of microservices, a term whose hype may have the ability to obscure its meaning. Blurred lines between architecture styles like SOA and microservices have caused some disparities in what developers specifically consider as microservices. Furthermore, it's becoming more and more expected for applications to be built in less monolithic ways, as monoliths can negatively affect scalability, time-to-release, and more. Microservices, therefore, in an age of continuous delivery and integration, are beginning to represent a new status quo for software development.

Still, while the popularity of microservices rises, their use is not universal. Of six integration architecture styles/patterns we asked about in our survey, microservices was only the fourth most popular. MVC and similar architectures, as well as layered (n-tier) and event-driven architectures, were all used by 20-25% more respondents. While these architectural styles obviously can't be compared side-by-side for every use case, this predicts that microservices, while quite useful for a number of situations, will

WHAT ADDITIONAL FUNCTIONALITY DO YOU NEED IN AN ENTERPRISE INTEGRATION SOLUTION?



WHAT SOFTWARE ARCHITECTURE PATTERNS AND STYLES DO YOU USE?



never provide a "silver bullet" solution to software architecture problems. Also, smaller teams or organizations are less likely to use microservices, either because they lack the need or the means to effectively use them. Companies with fewer than 500 employees (about 50% of our respondents) are 8% less likely to use microservices, while teams with fewer than 6 people (about 38% of our respondents) are 11% less likely to use microservices.

3. TOOLS CAN HELP, BUT THEY WON'T FIX EVERYTHING

80% of survey respondents said they use tools for API management, API design and documentation, and/or service registry and discovery. Respondents using any of these tools were about 8% less likely to say that their organization had significant difficulties with poor documentation and unclear requirements, the two most common integration difficulties mentioned in this year's survey, than those not using any of these tools. Users of these tools, however, were more likely to complain of other integration difficulties, such as propagating changes to integrated systems or dealing with different standards interpretations. This seems to denote that difficulties integrating systems won't be eliminated easily. Even as tools come along to help with some of the most common or most frustrating issues around integration, this may only bring new difficulties—or perhaps difficulties that were previously less concerning—to light. With so many moving parts involved in integrating such vastly various systems, it's no doubt that there will be plenty of hiccups along the way.

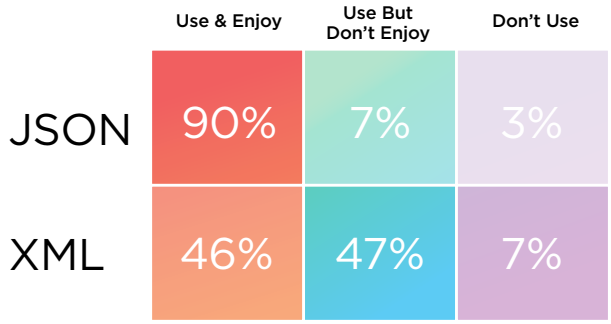
4. JSON PULLS AHEAD

Last year, XML and JSON were neck-and-neck as contenders for the most widely used data serialization and interchange format in integration, with XML just barely beating out JSON. This year, JSON has come out ahead, with 97% of respondents using it, as opposed to 93% who use XML. This is still within the margin of error for our survey results, and so it doesn't show clearly that JSON is being used more than XML for integration. However, an astounding 90% of respondents said that they not only use JSON, but they enjoy using it. XML users, on the other hand, were much more divided. Half of respondents saying that they use XML as a data serialization and interchange format said they do not enjoy it. If developer preference is any indicator, this points to JSON continuing to break down presuppositions that RESTful web services equate to XML over HTTP.

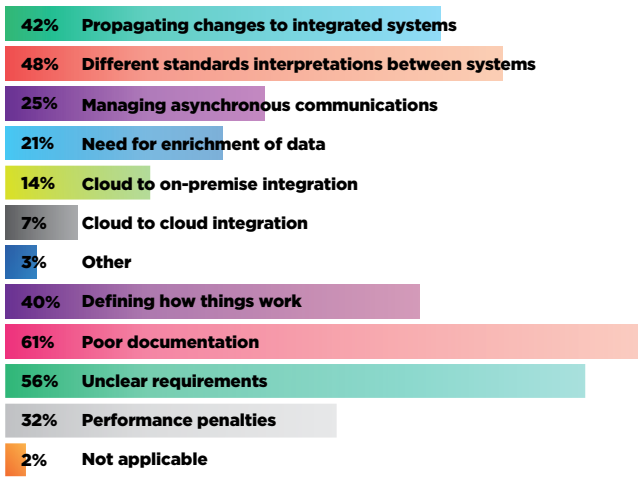
5. API VERSIONING HAS YET TO BE SOLVED

Methods for API versioning are abundant, and there has been plenty of debate regarding which method is best. In our survey, we asked about 8 different methods of explicit versioning, from HTTP codes to API gateways to versioning in the codebase. Results were, expectedly, mixed. The most popular method, used by 35% of respondents, was versioning in the endpoint URI without aliases. The next most common method was versioning in the endpoint URI with backward-compatible aliases—25% of respondents said they use this method. Perhaps most surprising, however, is that 24% of all respondents said they do not use explicit versioning for their REST APIs. This presents a symptom of the root of many great integration frustrations: a lack of enforced predetermined standards that define how systems are built and how they communicate. Ignoring best practices in software development is probably something we've all been guilty of at one point or another; but when integrating with external systems becomes involved, as is ever more often the case, these shortcuts or shortcomings begin to have impacts beyond our own code—impacts that we may never see, but that are real, nonetheless.

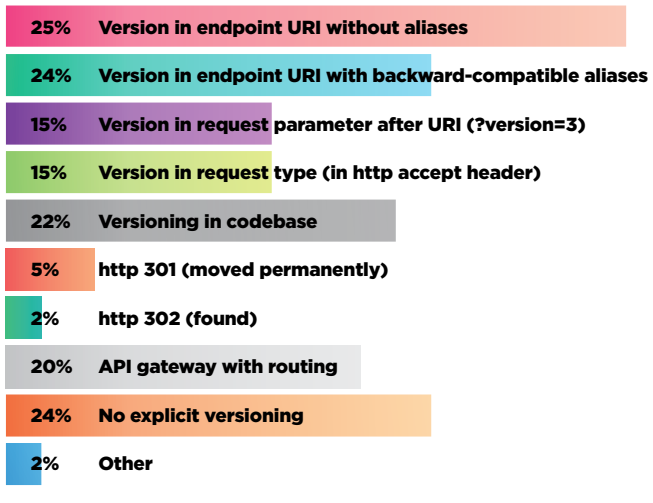
WHICH OF THE FOLLOWING DATA SERIALIZATION & INTERCHANGE FORMATS DO YOU (A) USE AND (B) ENJOY USING?



WHAT ARE YOUR ORGANIZATION'S BIGGEST INTEGRATION DIFFICULTIES?



HOW DO YOU VERSION YOUR REST APIs?



Cataloguing Microservices

BY **PETTER MÅHLÉN**
INFRASTRUCTURE ENGINEER AT **SPOTIFY**

QUICK VIEW

01 Microservices allow you to scale your development organization, which allows you to build more microservices. This leads to a need for systematic cataloguing and management.

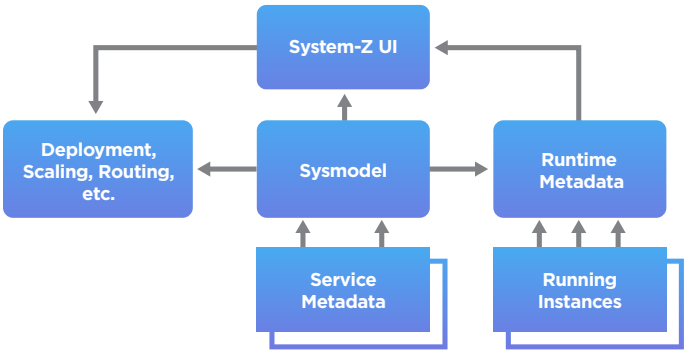
02 A great cataloguing system can form the foundation of other tooling, further helping the organization's productivity.

03 The self-reinforcing combination of the above two points enables your organization to experiment more and learn faster.

There is no such thing as a free lunch; almost every technology or architecture choice we make comes with pros and cons. Martin Fowler and his colleagues at ThoughtWorks have written an excellent article about the tradeoffs involved with using a microservices architecture. This article goes into some detail about one particular issue that is not covered there, using Spotify's software cataloguing system as an example.

One of the benefits of a microservices architecture is that it helps you scale your development organization, through a combination of strong module boundaries and independent deployment. So you can have more teams build services more quickly. This, like so many good things in computer science, is a double-edged sword, since the ability to build many services quickly tends to lead to… many services. When you have a large ecosystem of many small services, even if each service is simple, the sheer volume means that understanding the ecosystem becomes hard.

At Spotify, we're currently around 100 teams that independently build, deploy, and run microservices in our backend. We've got nearly 1600 services in our catalogue and about 1000 names are registered in our service discovery systems (see below for more on the difference). This means we're long past the point where word-of-mouth is enough to find who owns a service and what it does. We're on the second generation of tracking tools, using an in-house tool called System-Z to catalogue our software.



SYSTEM-Z: A SOFTWARE CATALOGUING AND TOOLING SYSTEM

System-Z consists of a set of microservices (obviously!) and a web UI (see top right).

At the heart of System-Z is a service called "sysmodel," which tracks statically configured metadata about the various microservices in our ecosystem. This information has the same kind of problem as any documentation: the person or team who is capable of providing it is not the person or team who most benefits from it being in great shape. Since it is beneficial for Spotify as a whole to have great quality metadata about our services, we try to encourage teams to keep their service metadata up to date, through, for instance:

1. Storing the metadata together with the code, clearly highlighting that the metadata has the same owner as the code.
2. Making it easier to use tools for managing your services if the data is good.

3. Showing warnings and hints that make your services look "untidy" if the data isn't up to date, hopefully appealing to engineers' sense of cleanliness to fix the warnings.

In order to get some data that is dynamic by nature (where is this thing running, which version is it, etc.), and to improve the reliability of some data (which other services is it actually calling, etc.), we also collect runtime data by polling the running instances. The metadata is created and published by our backend framework Apollo.

System-Z has also become the place where most of the tools for managing backend services are made available, and most of those tools tend to be built on top of the data that the sysmodel service provides.

THE MODEL

Some of the core concepts in the sysmodel data model are:

1. **Software Component:** although System-Z and sysmodel were developed to support our microservices, we track not only microservices here, but also data processing pipelines, libraries, third-party tools like Jenkins, etc.

2. **Role**: a function that we want to scale to a certain amount of users, or deploy to a certain number of availability zones, to specific geographic locations, etc. An example could be 'login', which is what allows Spotify users to log in. A Role is usually instantiated on a (virtual) host and requires one or more components to be running on the host in order to work. Roles are typically scaled horizontally to a sufficient number of hosts. A Kubernetes Pod is a good example of an implementation of this concept.

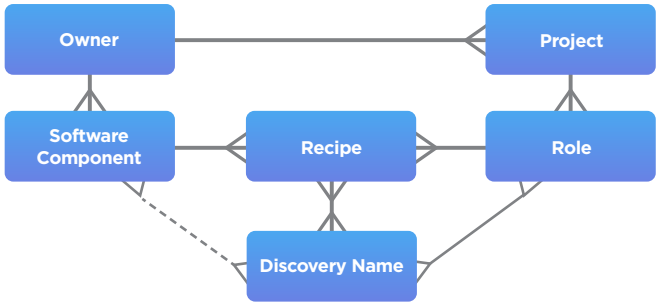
3. **Project**: a set of roles that are related; could for instance be the 'login' role together with the data store that contains user data.

4. **Recipe**: a description of the software components needed to be installed together on a host that plays a role. A Kubernetes pod template is a good example of an implementation.

5. **Discovery name**: to express dependencies between microservices, we use what we call a discovery name. This indirection allows us to do things like inserting a proxy in front of an existing service, for caching, upgrading, deprecation or dark launching purposes, helping us run and improve our product without downtime. A deployed component can register zero or more discovery names.

6. **Owner**: one of the most frequent questions about software in our catalogue is 'who owns it', as knowing that is necessary to understand who to ask 'how do I use it to do X?'.

The actual format of the data the sysmodel service reads is free-form YAML, meaning that users are free to add their own service metadata if desired. We decided to expose the



sysmodel service publicly within Spotify when we launched it in May 2015. A review in September 2016 of what people internally are using it for found no less than 18 different use cases, from business rules for server access control ("if you're a member of a team owning service X, you get login rights to server Y") to automatically updating all monitoring dashboards for services owned by a particular team. Most of the use cases people found for the data served by sysmodel were things we had not anticipated when building it.

CONCLUSIONS

A great thing about the freedom that a microservices architecture gives to teams and individuals is that it allows the decentralized creation of many small components. This improves the speed of experimentation and learning, but leads to an increased volume of software, which in turn makes it hard to understand the software ecosystem: what should be out there, who owns it, and how does it fit into the bigger picture? Can I deprecate it? A microservice catalogue like System-Z simplifies this understanding, and also serves as a great base for other tooling that works across the board with your backend systems.

What's more, people and teams at Spotify are right now creating about 20 new services per week. Most of these are either for learning how to build a backend service or experiments and will never make it to production. This number has grown by leaps as we've made changes that make it easier for teams to build and deploy a service to production. We believe that being great at learning quickly is a strategic advantage for us, so this is a pleasing development.

We've still got a number of things we want to improve in System-Z. Number one is probably making it (or most of it) open-source. Another is API discovery; being able to go from use case to a service API that solves your use case. Despite its shortcomings, it does make many things easier that used to be hard, and makes some things possible that were impossible before.

PETTER MÅHLÉN Currently building infrastructure and developer tools at Spotify, Petter has 20 years of experience of software development in many roles: developer, project manager, product owner, CTO, architect, etc. The last few years, he has eschewed management roles for love of coding, and has worked mostly on building large-scale distributed systems for the web.

REST API Basic Guidelines: Design it Right

BY **GUY LEVIN**
CTO OF **RESTCASE**

QUICK VIEW

01 REST constraints help build simple, visible, accessible, evolvable, flexible, reliable, maintainable, scalable, and performant services.

02 When designing and developing a RESTful API, consider building microservices.

03 Start your RESTful API design with thinking about versioning, it will benefit clients and servers (services).

Your data model has started to stabilize and you're in a position to create a public API for your web app or device.

You realize it's hard to make significant changes to your API once it's released and want to get as much right as possible up front. Now, the internet has no shortage on opinions on API design.

Good REST API design is extremely hard, especially when considering that an API actually represents a contract between you and those who consume your data. Breaking this contract may have a very bad impact starting with time being wasted in development and ending with many angry users with mobile apps or services that no longer work.

But, since there's no one widely adopted standard that works in all cases, you're left with a bunch of choices: What formats should you accept? How should you authenticate? Should your API be versioned?

Let's first start with the basic guidelines...

WHAT IS REST?

Representational State Transfer (REST) is a technical description of how the World Wide Web works. If you imagine that the Web is a device, and it could have an operating system, its architectural style would be REST.

REST defines a set of architectural principles by which one can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model. In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface design because it's a considerably simpler style to use.

REST CONSTRAINTS

A well-designed REST API entices developers to use the web service and is today a must-have feature. But how do you clearly define that the API is actually RESTful? REST architecture describes **six constraints**, and we are going to describe them below in this article.

By conforming web applications, web services, and web APIs with proven REST constraints, teams can create very scalable, simple, maintainable, evolvable, and reliable systems.

Here are the REST six constraints:

1. Uniform Interface
2. Client Server
3. Stateless
4. Cacheable
5. Microservices
6. Code-on-demand

Each constraint adds beneficial properties to the web system. By incorporating the constraints, teams can build simple, visible, usable, accessible, evolvable, flexible, maintainable, reliable, scalable, and performant systems.

BY FOLLOWING THIS CONSTRAINT	GAIN THIS SYSTEM PROPERTY
Uniform Interface	Simple, usable, evolvable, and reliable
Client Server	Simple, evolvable, scalable
Stateless	Simple, maintainable, evolvable, and reliable
Cacheable	Scalable, and performant
Microservices	Flexible, scalable, maintainable, reliable, and performant
Code on demand	Evolvable

Table 1: How following specific REST constraints will result in gaining valuable system properties.

UNIFORM INTERFACE

The uniform interface that any REST service must provide is fundamental to its design. Its constraint defines the interface between clients and servers. The four guiding principles of the uniform interface are:

- **Resource-Based**: Individual resources are defined in requests using URIs as resource identifiers and are separate from the responses that are returned to the client. REST Web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.

- **Actions on Resources Through Representations**: When a client gets a representation of a resource, including any metadata attached, it has enough information to customize or delete the resource on the server, if it has permission to do so. REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:
  - To create a resource on the server, use POST.
  - To retrieve a resource, use GET.
  - To change the state of a resource or to update it, use PUT.
  - To remove or delete a resource, use DELETE.

- **Self-descriptive Messages**: Each client request and server response is a message, and RESTful applications expect each message to be self-descriptive. That means each message contains all the information necessary to complete the task. Other ways to describe this type of message is "state-less" or "context-free." Each message passed between client and server can have a body (or "entity body") and metadata. RESTful applications also operate on the notion of a constrained set of message types that are fully understood by both client and server. There are well-defined rules for how clients and servers are expected to behave when using these messages. The names and meanings of the messages' metadata elements (HTTP Headers) are also well-defined.

- **Hypermedia as the Engine of Application State (**HATEOAS**)**: Clients deliver the state via body contents, query-string parameters, request headers, and the requested URI. Services deliver state to clients via body content, response codes, and response headers. A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses.

CLIENT SERVER

The uniform interface divides clients from servers. This means that, for instance, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not engaged with the user interface or user state so they can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not modified.

STATELESS

There is no connection state. Interaction is stateless. Each new request should carry all the information required to complete it, and must not rely on previous interactions with the same client. The necessary state to operate the request is contained within it as a part of the URI, query-string parameters, body, or headers. The

URI identifies the resource, and the body contains the state of it.

CACHEABLE

Resources should be cacheable whenever possible (with an expiration date/time). As the clients can cache responses, the protocol must allow the server to explicitly specify which resources may be cached, and for how long, in order to prevent clients from reusing state or inappropriate data in response to further requests:

- Since HTTP is universally used as the REST protocol, the HTTP cache-control headers are used for this purpose.
- Clients must respect the server's cache specification for each resource

Well-managed caching partially or completely eliminates some client–server interactions and improves the performance.

MICROSERVICES

There are different definitions of microservices, and searching the Internet provides many good resources that provide their own viewpoints and definitions. However, most of the following characteristics of microservices are widely agreed upon:

- Encapsulates a customer or business scenario. What is the problem you are solving?
- Developed by a small engineering team.
- Written in any programming language and uses any framework.
- Consists of code and (optionally) state, both of which are independently versioned, deployed, and scaled.
- Interacts with other microservices over well-defined interfaces and protocols.
- Has unique names (URLs) used to resolve their location.
- Remains consistent and available in the presence of failures.

Microservices are applications composed of small, independently versioned, and scalable customer-focused services that communicate with each other over standard protocols with well-defined interfaces.

CODE ON DEMAND

Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript. This is the only constraint out of six that is optional.

VERSIONING

No matter what you are building, no matter how much planning you do beforehand, your core application is going to change, your data relationships will change, attributes will invariably be added and removed from your Resources. This is just how software development works, and is especially true if your project is alive and used by many people (which is likely the case if you're building an API).

Getting RESTful API versioning right can have a major impact on the how your API is perceived by your API consumers internally or externally, and can also make the management of your API estate more difficult if it's ill conceived.

CONCLUSION

When starting to design and develop a RESTful API, you first must learn the constraints REST is enforces to the system you are building. Violating any constraint other than Code on Demand means that service is not strictly RESTful. Complying with all constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as **performance, scalability, simplicity, modifiability, visibility, portability, and reliability**.

Getting RESTful API versioning right can have a major impact on how your API is perceived by your API consumers internally or externally. Being aligned with the REST constraints will also help you with your API versioning both in your development when using the microservices constraint and in your design when creating the uniform interface constraint.

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

CHECKLIST FOR

Hypermedia APIs

"REST IS EASY FOR THE CONSUMER BUT COMPLEX TO IMPLEMENT FOR PROVIDERS"

When planning to expose an API to the user, you should always think about who is going to use this API. The clients of hypermedia APIs include other developers, so developers of hypermedia APIs must provide the following features to these clients.

1. A simple and understandable URL
2. A proper response after the client invokes a URL
3. An easy method of debugging

Below, we'll cover the Do's and Don'ts for the following sections:

- Resource URI Design and Resource Representation
- HTTP Methods
- Response Codes
- HATEOAS

URI DESIGN & RESOURCE REPRESENTATION

DO'S:

1. Always use a noun for resources. For example: Articles, Refcardz, Blogs, etc.
2. Use plural nouns for resource names to represent collections. For example: /refcardz, /articles, etc.
3. Use an identifier after the resource name to specify an instance. For example: /refcardz/{id}, /articles/{id}, etc.
4. Use camelCase for Resource names like /researchGuides
5. To optimize searches, always use paginated views of the collection of results and provides hypermedia links for the page number. For example: /refcardz should return 10 or 20 Refcardz per request rather than returning all Refcardz.
6. Always provide an efficient search mechanism by providing a filter using the "?" and "&" query parameters. For example: /articles?category=java&year=2016
7. Enable sorting by a query parameter. For example: /articles?category=java&sortBy=publishDate
8. Choose a suitable content type and charset as header parameters (define media type) for response (e.g. "application/json;charset=utf-8" for a JSON response.
9. Specify the content language to record the language used.
10. Always use versioning in your API URL's (e.g. /v1/articles, /v1/refcardz).

DON'TS:

1. Don't use a verb for resources. For example: /getRefcardz. /getArticles.
2. Don't use many URLs based on filter criteria: this actually confuses clients. For example: /getArtclesByAuthorName, /getArticlesByCategory, /getArticlesByValuableAutor, etc.
3. Don't use a trailing forward slash. For example: /refcardz/, /articles/{id}/, etc.
4. Don't use filename extensions for content types. For example: /refcardz.json, /articles.xml, etc. (Rather use a suitable content type header.)
5. Don't send excessive data in a response so that you can avoid creating network blockage or a heavy response.

HTTP METHODS

DO'S:

OPERATION	HTTP METHODS	RESOURCE URL
Create an Article	POST	/articles
Delete an Article	DELETE	/articles/{id}
Get a specific Article	GET	/article/{id}
Search for all articles	GET	/articles
Update a Specific Article	PUT	/articles/{id}

1. For GET, always use a cache control header to explicitly specify if the response is cached or not.
2. Always use POST to add a new element to collection resources. For example: /article/{id}.
3. Always use a location header in the response to tell the client about the newly created resource so that they can clearly understand what location the new resource has been created in.

DON'TS:

1. Don't use the PUT operation for creating resources.
2. Don't use a query string with POST or PUT as input parameters. For example, POST / articles?category="java"
3. Don't use POST for an update.
4. Don't allow delete on collections.

ABOUT THE AUTHOR

SHAMIK MITRA has ten years of experience working with Java. Currently, he is working for IBM as a Technical Leader. He's a self-proclaimed Java maniac that loves to share his knowledge about Java and new IT trends.

HATEOAS

DO'S:

1. Always provide suggestive links to help navigate to related resources. In this example, we'll show how to navigate from Article to Author:

```
{
    "Article": "HyperMedia API Checklist",
    "links": \[ {
        "rel": "Author",
        "href": "http://dzone.com/author/1"
    } \]
}
```

2. Make sure your API is explorable by providing enough suggestive links.

DON'TS:

1. Inability to maintain HATEOAS will incur failure in achieving the **third level of the Richardson Maturity Model.**

RESPONSE CODE

DO'S:

1. Always use the appropriate response code for the operation. Go to the link restapitutorial.com/httpstatuscodes.html for more info.

DON'TS:

1. Don't use a 500 internal error code for an incorrect client request. For example, one that's missing a mandatory header, has malformed JSON, etc.

IBM

IBM **API** Connect

API Connect integrates IBM StrongLoop and IBM API Management with a built-in gateway, allowing you to create, run, manage, and secure APIs and Microservices.

Unparalleled, integrated user experience.

ibm.biz/apiconnect

Strangle Your API:

Grow a New API Around Your Current Code

If you and your company were early riders on the API bandwagon it is more than likely that now you're looking to update, upgrade, and upsell your current customers to new functionality and new services. You may be looking longingly at decomposing a large service into microservices, experimenting with FAAS/serverless architectures, or even dipping a toe into hypermedia. And you likely have a wish list of new features, new functionality, and bug fixes to attend to, as well.

Updating an in-production API can be a daunting task, but luckily there's a pattern to help: the strangler pattern. Martin Fowler named this pattern in a blog post in 2004—on a trip to Australia, he'd noticed strangler figs which began growing in the upper branches of a tree and worked their way over time down to the soil, eventually completely enclosing (and killing!) their host tree. He realized you could replace an important system the same way: starting at the branches, and working your way back towards the root.

Why choose a strangler pattern over a complete rewrite? Fowler suggests the best reason to do so is to manage risk: you can implement a strangler pattern in small bites and release frequently.

It's also easier than ever to implement a strangler pattern today, especially for an API.

Using an API gateway, load balancer, or reverse proxy (e.g. NGINX), you can capture and intercept calls to your old API, and decide whether they go to your legacy code or to new code you have written—perhaps even depending on the query parameters—letting you support new features in situ and gracefully sunset old functionality.

You can also strangle your existing APIs lower in the stack: for example, with a framework such as LoopBack, you can wrap existing APIs in new APIs, modifying the data that is returned.

If you're looking for case studies and a more in-depth discussion, check out this blog post by Paul Hammant.

WRITTEN BY ERIN MCKEAN
LEAD STRONGLOOP & NODE.JS DEVELOPER EVANGELIST, **IBM**

PARTNER SPOTLIGHT

API Connect By StrongLoop and IBM

IBM API Connect is a complete solution that addresses all aspects of the API lifecycle—Create, Run, Manage, Secure—for both on-premises and cloud environments.

CATEGORY
API Management

NEW RELEASES
As needed

OPEN SOURCE
No

CASE STUDY

Create—create high-quality, scalable, and secure APIs for application servers, databases, enterprise service buses (ESB), and mainframes in minutes.

Run—take advantage of integrated tooling to build, debug, and deploy APIs and microservices using Node.js or Java.

Manage—create and manage portals that allow developers to quickly discover and consume APIs and securely access enterprise data, and monitor APIs to improve performance.

Secure—manage security and governance over APIs and microservices. IT can set and enforce API policies to secure backend information assets and comply with governance and regulatory mandates.

STRENGTHS

- Simplify discovery of enterprise systems of record for automated API creation
- Provide self-service access for internal and third-party developers through a market-leading gateway
- Ensure security and governance across the API lifecycle
- Unify management of Node.js and Java microservice applications
- Increase flexibility with hybrid cloud deployment

FEATURES

- Unified Console
- Quickly run APIs and microservices
- Manage APIs with ease
- Readily secure APIs and microservices
- Create APIs in minutes

BLOG developer.ibm.com/answers/topics/apim

TWITTER @ibmapiconnect

WEBSITE ibm.com/software/products/en/api-connect

Transactions for the REST of Us

BY **GUY PARDON** FOUNDER OF **ATOMIKOS**
AND **CESARE PAUTASSO** ASSOCIATE PROFESSOR AT **USI LUGANO, FACULTY OF INFORMATICS**

QUICK VIEW

01 You need to perform multiple POST requests over different RESTful APIs and make sure that either all of them happened successfully or none of them did.

02 You can model each state transition triggered by the POST request as a temporary state transition which can either be confirmed to become permanent or canceled (if no confirmation arrives before it expires) and thus get reverted back to the initial state.

03 Then you can use the TCC protocol we illustrate in this article to ensure atomicity of the distributed transaction across multiple REST APIs.

Does REST need transactions? In this article we take a pragmatic approach driven by concrete examples, which can benefit from atomic transactions across REST services. We then show how Try-Confirm/Cancel (TCC) can offer a simple, interoperable solution that is aligned with REST/HTTP and Hypermedia. TCC can be universally applied because it is based on a design pattern rather than a product or technology. For a specific but important class of systems—reservation systems—it provides both a transaction model as well as a lightweight form of "BPM" over REST.

REST SYSTEMS THAT BENEFIT FROM DISTRIBUTED ATOMIC TRANSACTIONS

We claim that a whole class of systems termed "distributed reservation systems"—even if implemented in REST—benefit from transactions. The following two examples will show you why:

1. Travel: Booking a flight and a rental car together at two independent providers. Suppose you want to book a flight to Barcelona and a rental car to drive to the South of Spain. You don't want a car without the flight, and you don't need a flight if there is no car available.

2. Telco: Reserving a custom phone number for a customer—subject to billing conditions. A telco company allows selected customers to purchase customized phone numbers. The phone number is checked for availability and, if available, reserved for the customer. Then a separate billing service is called for the billing process. When billing succeeds, the number is assigned to the customer. If billing fails, then the phone number is released again for the next customer. It would be bad for the telco if it assigned phone numbers without billing the customer or if it billed customers for phone numbers not assigned.

As a generalization, we would like to have reservation processes across multiple, independent REST services where failures before completion are guaranteed to have no effect. This is similar to the classic well-known transactional programming model, where transactions roll back if they do not reach the commit stage.

OUR SOLUTION: TRY-CONFIRM/CANCEL (TCC)

We offer a solution for distributed atomic transactions across REST services that aligns well with HTTP and REST principles, and offers the simplest thing that could possibly work. We promised to be pragmatic, so let's define our solution by its desired behavior in terms of, say, the travel example:

THE HAPPY PATH

We want the travel example's "happy path" to work like this:

- **Step 1:** Book the flight at the airline provider via HTTP POST. The airline gives back a URI to confirm and an expiration deadline for confirmation.
- **Step 2:** Rent the car at the car rental company via HTTP POST. The car rental company returns a URI to confirm and an expiration deadline for confirmation.
- **Step 3:** Confirm both previous steps by calling PUT on each of the URIs returned.

EITHER CANCEL EVERYWHERE, OR CONFIRM EVERYWHERE

The airline and/or the car rental company are free to autonomously cancel their bookings after the expiration deadline. This means that any failures before step 3 (i.e., not executing step 3) imply that neither the airline provider nor the car rental company receive a confirmation before their respective expiration deadlines, after which they are free to cancel autonomously—thereby releasing the business resources they have been reserving until then. The net result: cancel everywhere (after some time).

Any failures after step 3 do not affect the atomicity of the outcome, since both the flight and the car have already been confirmed, so both reservations have been successful.

THE TRICKY PART

Failures during step 3 are trickier: temporary outages of either participant can be resolved by resending the confirmation message. After all, PUT is idempotent, so we can keep retrying it as long as necessary. However, as soon as the airline provider has been confirmed, confirmation to the car rental may still fail (for instance, due to intermediate expiration followed by autonomous cancel). This is always possible and would lead to a "heuristic" anomaly—just like you will find in any distributed transaction system ever built. To minimize the occurrence of heuristics and to facilitate their resolution, smart implementations can be based on a specialized and reusable TCC coordinator that makes informed decisions about whether to proceed in step 3 or not, keeps a recoverable progress log and performs smart retries where possible. This way, the number of anomalies may be reduced to practically zero.

IMPLEMENTING TCC

TCC only works if each of the roles (participant service, application, and coordinator) follow their part of the protocol. While this may seem like a constraint, in reality the protocol is so simple that it can be easily achieved in many application domains.

THE TCC SERVICE / PARTICIPANT

TCC participants (like the airline or the car rental) implement a simple lifecycle model for service invocations ("reservations"). Each invocation of a participant service goes through the following three states:

- **Initial**: Nothing has been done yet.
- **Reserved**: An invocation ("Try")—probably via HTTP POST with a local database transaction within the service—has lead to a reservation on behalf of the client process. The reserved state is identified by a unique URI that can be used to confirm, and an associated expiration date/time after which the participant can autonomously cancel and move back to the initial state. In the case of a flight reservation system, this corresponds to a seat reservation identified by a URI and with associated expiration date/time. As a minor but useful extension, a participant in this state can accept HTTP DELETE in case it wants to be notified of failures before it times out by itself.
- **Final:** This is where the reservation becomes permanent, meaning it can no longer be easily and automatically cancelled. This state is reached when a reserved participant receives a confirmation message within the specified time frame, by HTTP PUT.

  For reporting purposes, the participant service's database should reflect these three states. In particular: the distinction between reserved and final state needs to be clear for each reservation because:

  1. Autonomous cancel is only allowed during reserved state.
  2. Sales reporting probably only wants to take the final reservations into account.
  3. Penalties may be levied to clients that perform cancellations on finalized bookings, but not on temporary reservations.

THE TCC APPLICATION / WORKFLOW

The application or workflow logic is according to the happy path outlined above. In particular, no specific error path logic is needed—which simplifies the logic and makes it more reliable by taking out the hard-to-test parts. Either step 3 is reached (implying confirmation), or not (implying cancellation).

THE TCC CONFIRMATION / COORDINATOR

When the happy path reaches step 3, the TCC coordinator service comes into play: it accepts the set of URIs to confirm and tries to do a smart job by confirming and retrying them if needed, or cancelling when that is the better option—at least for those participants that accept DELETE.

A RESTFUL SOLUTION

TCC fits really well with REST. Every participant (the car reservation company and the airline) always returns a URI pointing to the reserved resource. This uses hypermedia so that clients can inspect the URI with GET to inquire about its validity (when is the reservation going to expire?) but the URI can also be used to confirm the reservation by sending idempotent PUT requests. We have defined a specific MIME type for TCC participants to facilitate compatibility between participants, applications, and the coordinator. This means you can define participants today and have them participate in the transactions of tomorrow.

REFERENCES

- For a more elaborate discussion of TCC for REST, see this recorded presentation: infoq.com/presentations/Transactions-HTTP-REST
- For details on the coordinator API and the minimum requirements for the participants API see: Guy Pardon, and Cesare Pautasso, "Atomic Distributed Transactions: a RESTful Design", *Proc. of the 5th International Workshop on Web APIs and RESTful Design (WS-REST)*, Seoul, Korea, ACM, April, 2014. dx.doi.org/10.1145/2567948.2579221
- The detailed TCC API specification (including how to implement your own participant) is available here: atomikos.com/Blog/TransactionsForRestApiDocs?Source=dzone

GUY PARDON @guypardon is the original founder of Atomikos, a company offering reliability solutions such as TCC for REST. He coordinates the technical and support efforts of the team. Guy is also the chief software architect in charge of product design and manages the open source community at Atomikos.

CESARE PAUTASSO @pautasso is associate professor at the USI Faculty of Informatics in Lugano, Switzerland. In 2012 he co-authored a book on SOA with REST, while he's currently writing a book titled "Just send an email: Anti-Patterns for Email-Centric Organizations" published on Leanpub, leanpub.com/email-antipatterns. He recently worked on a set of RESTful conversation patterns available on: restalk-patterns.org.

INTEGRATION STATION

Distributed systems, microservices architecture, SOA; all of these are becoming more and more common, and necessitate the use of integration solutions. However, with so many tools out there, which ones are actually seeing developer adoption?

We asked over 700 DZone members about which integration tools and frameworks they use to communicate between their application components. We looked at their top choices and examined some of the benefits and shortcomings of each. Get your ticket to ride and take a look:

NOTHING

1. 30.2% of survey respondents may not be using an integration solution, but 8.9% are currently looking for one.
2. When asked what they wish they used, the top four tools were:
   Spring Integration: 12.0%
   Camel: 8.7%
   Mule ESB: 6.0%
   Jenkins: 2.7%
3. 68.9% of integrations are built ad hoc, rather than by using a tool.

30% ARE NOT USING AN INTEGRATION SOLUTION

SPRING INTEGRATION

39% OF SURVEY RESPONDENTS ARE USING SPRING INTEGRATION

1. Primarily used for messaging and scheduling. Can also fetch data, process it, and write it
2. Lets you build apps with event-driven architectures.
3. Uses XML to write integrations, but offers no DSL.

APACHE CAMEL

28% OF SURVEY RESPONDENTS USE CAMEL

1. Used to route and mediate messages between components
2. Uses URIs to work directly with any kind of messaging model, and uses pluggable components and data format options.
3. Uses an XML-based DSL, but also offers DSLs for Java, Groovy, and Scala.

ORACLE ESB

13% OF RESPONDENTS USE ORACLE

1. Designed to work in SOA environments using both Oracle and third-party software
2. Includes built-in monitoring, management, and Quality of Service tools.
3. Packaged with the JDeveloper IDE.

JBOSS FUSE ESB

11% OF RESPONDENTS USE JBOSS

1. SOA infrastructure that integrates components by using standardized methodologies and tools.
2. Can be used to create and connect APIs together, and to debug integration
3. Packaged with Apache Camel.

A Survey of Modern Application Integration Architectures

BY **KAI WÄHNER**
TECHNOLOGY EVANGELIST AT **TIBCO SOFTWARE**

QUICK VIEW

01 One core integration platform is not sufficient in the era of cloud, mobile, and IoT.

02 Various integration options such as on-premise application integration, iPaaS, iSaaS, or process integration are relevant for different audiences, like the integration specialist, ad hoc integrator, or citizen integration (aka business user).

03 Different integration options have to be loosely coupled but highly integrated.

04 Open API intiatives and API Management tools should be leveraged to expose internal services to other departments, partners, or public developers.

The IT world is moving forward fast. Cloud services, mobile devices, and the Internet of Things peel away existing business models, but also establish wild spaghetti architectures through different departments and lines of business. Several different concepts, technologies, and deployment options are used. A single integration backbone is not sufficient anymore in this era of integration.

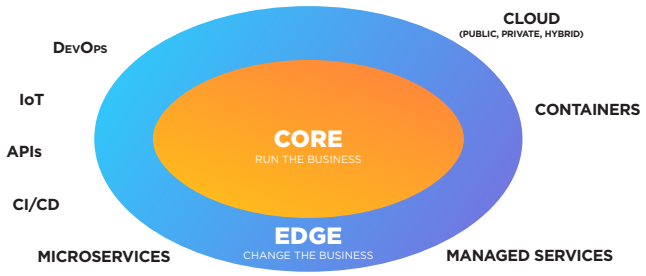
A HYBRID INTEGRATION PLATFORM FOR CORE AND EDGE SERVICES

"Hybrid Integration Platform (HIP)" is a term coined by Gartner and other analysts. It describes different components of a modern integration architecture.

Leveraging a well-conceived hybrid integration architecture allows different stakeholders of an enterprise to react quickly to new requirements. Mission-critical core business processes (also called "core services") are still operated by the central IT department. These services **run the business** and change rather infrequently.

On the other side is the line of business needing to try out new—or adapt existing—business processes quickly in an agile way without the use of delaying and frustrating rules and processes governed by central IT. Innovation by a "fail-fast" strategy and creating so-called "edge services" is getting more and more important to enhance or disrupt existing business models and therefore **change the business**.

Figure 1: Enterprise Journey with CORE and EDGE Services



The following figure shows the components needed in a Hybrid Integration Platform to run **and** change the business successfully:

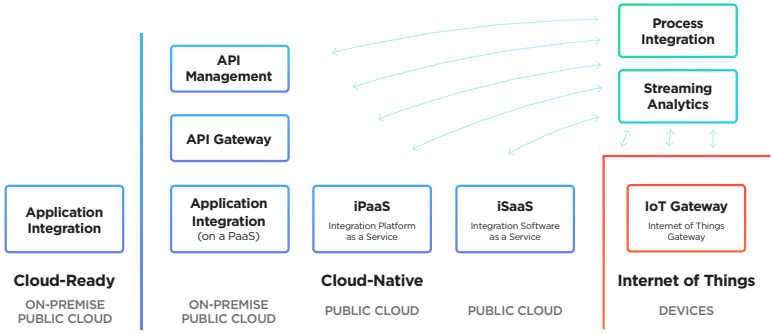


Figure 2: Components of a Hybrid Integration Platform

APPLICATION INTEGRATION WITH AN ENTERPRISE SERVICE BUS (ESB)

WHEN TO USE IT?

ESBs are usually used in bigger IT projects where **mission-critical core business processes** have to be integrated with

a need for high availability, reliability, and performance within the enterprise ("**core services**"). This affects many different technologies and applications to integrate standard software, legacy systems, custom applications, or external cloud services.

DEPLOYMENT MODEL AND TARGET AUDIENCE

Integration specialists implement ESB clusters to leverage high performance, high availability, fault tolerance, and guaranteed delivery of transactions. Most of the integration happens on premise in the private data center. Therefore, even "Microservices do not spell the end of the ESB!" You can deploy to cloud, but this is more "cloud-washed"—but not cloud-native—deployment and operations; you "just deploy your existing application to the cloud!" However, this also can make sense, like for test instances or to use IaaS capabilities without leveraging cloud-native features.

SOME OPTIONS?

An Enterprise Service Bus (ESB) includes products such as MuleSoft Anypoint Platform, Talend ESB, WSO2 ESB, TIBCO ActiveMatrix BusinessWorks, or Oracle Service Bus. An alternative pure-coding solution is using an open-source Integration Framework such as Apache Camel or Spring Integration.

CLOUD-NATIVE APPLICATION INTEGRATION

WHEN TO USE IT?

Integration in a cloud-native environment is quite different than your traditional approach to application integration. Typically, companies leverage platforms-as-a-service (PaaS) such as Cloud Foundry, Kubernetes, OpenShift, or Apache Mesos.

The **applications created using these cloud-native application integration tools run natively within these PaaS environments** and therefore benefit from the features offered by these platforms, including provisioning infrastructure, service discovery, load balancing, elasticity, cluster management, or fail-over out-of-the-box. It also allows more agile development to implement, adopt, and scale new features or innovations quickly and efficiently.

For this, the application development and architecture needs to be adapted to cloud-native concepts. This especially includes applying concepts behind "The Twelve-Factor App" principles, which recommends best practices for cloud-native applications such as stateless services, automation via DevOps, or environment-independent backing services. Often, you leverage independent containers (e.g. Docker, CoreOS, or Cloud Foundry's Warden) for building cloud native microservices or applications.

DEPLOYMENT MODEL AND TARGET AUDIENCE

Core IT teams are adopting these PaaS platforms, but they are doing this to drive the agility for **all** developers. As a result, both the traditional integration teams as well as the developers within line of business are able to benefit from these platforms and the capabilities offered by the integration technologies.

PaaS are widely adopted: on premise, in the public cloud, or as hybrid deployments. However, many enterprises do not have a complete, long-term strategy yet regarding cloud or hybrid deployment architectures. Hence it is important to **develop cloud-platform-agnostic integration services**, which can be moved from one platform to another without great effort or redevelopment.

Middleware also should support and integrate mature, open-source frameworks for cloud native environments such as Spring Cloud Configuration, Consul, Netflix's Eureka, or Hystrix instead of re-inventing the wheel by introducing additional complexity. The article "A Cloud-Native Architecture for Middleware" explains the concepts behind cloud-native platforms and deployment options in much more detail.

SOME OPTIONS?

TIBCO BusinessWorks Container Edition or WSO2 are examples for cloud-agnostic integration middleware supporting different PaaS and container platforms such as Cloud Foundry, Docker, Kubernetes, or AWS ECS. JBoss Middleware Services allows the deployment of its middleware applications onto OpenShift.

INTEGRATION PLATFORM AS A SERVICE (IPAAS)

WHEN TO USE IT?

An iPaaS Cloud Integration middleware is a pure **public cloud offering hosted by a specific vendor**. Using iPaaS **allows the line of business to react quickly to new requirements or innovative ideas without struggling with the core IT team** and its long release and quality management processes. iPaaS can be used for both mission-critical core services and innovative edge services.

DEPLOYMENT MODEL AND TARGET AUDIENCE

The vendor takes care of cloud-native features such as provisioning of infrastructure, elasticity, or multi-tenancy. This needs to be a real cloud-native enterprise-grade runtime and not just a "cloud-washed" offering. Otherwise, it is not possible to scale out quickly and easily while still keeping high demands regarding stability and resiliency of integration services.

The target audience for iPaaS is not necessarily the integration specialist with extensive technical experience. iPaaS also **allows colleagues with less technical expertise (sometimes called "ad-hoc integrators") to define, deploy, and monitor services and APIs** with its corresponding policies. Ad-hoc integrators often do not use the more powerful IDE but an **intuitive and simple-to-use web user interface**.

iPaaS has to work well together with other integration solutions. How different integration solutions work together depends from vendor to vendor. You should check if you need to redevelop existing services, if you can add changes via a web UI, and if different products work together at all (including commercial support).

SOME OPTIONS?

The iPaaS market is just emerging these days. Some examples are Dell Boomi, Informatica Cloud, MuleSoft Anypoint Platform, SnapLogic, Jitterbit, or TIBCO Cloud Integration.

INTEGRATION SOFTWARE AS A SERVICE (iSaaS)

WHEN TO USE IT?

iSaaS integrations serve **"edge services," which are not strategic and mission-critical for the enterprise—but very relevant for the specific business user**. For instance, a business user creates a daily automatic flow to synchronize data from a Google Sheet with Salesforce CRM. This removes the need to integrate these updates manually every day.

DEPLOYMENT MODEL AND TARGET AUDIENCE

iSaaS is hosted and operated by the vendor. In contrast to the above integration components, **iSaaS focuses on business users** (also called citizen integrator in this context). They can create basic integration flows for personal or department interests in a very intuitive web user interface without any technical knowledge.

SOME OPTIONS?

Examples for iSaaS solutions are SnapLogic, TIBCO Simplr, or IFTTT ("If This Then That").

IoT INTEGRATION GATEWAY

WHEN TO USE IT?

The Internet of Things (IoT) changes the role of edge integration. It raises several new challenges not relevant for classical application integration such as low bandwidth or non-reliable connectivity.

Here you need to integrate data directly on the edge devices, as not all data should be sent to the private data center or public cloud. For example, you might aggregate and filter sensor data from an assembly line in manufacturing, and only forward relevant information (such as alerts) to external interfaces. An **IoT Integration Gateway** interconnects all devices via various IoT Standards such as MQTT, CoAP, WebSockets, Bluetooth or RFID.

DEPLOYMENT MODEL AND TARGET AUDIENCE

Integration specialists use the IoT Integration Gateway to realize IoT edge integration. Development can be done via coding or intuitive web user interface and out-of-the-box connectivity to various IoT standards.

SOME OPTIONS?

You can use hardware gateways from vendors such as Intel, ARM, or Eurotech, or IoT gateway open-source frameworks such as IBM's NodeRED (implemented with Node.js using JavaScript) or TIBCO's Flogo (implemented with Google's Go Programming Language).

API MANAGEMENT

WHEN TO USE IT?

The trend is heading towards an "**Open API Economy**" where services are exposed as APIs to other internal departments, partners, or public developers. Think about examples such as PayPal, whose API is integrated into almost every online shop as a payment option, or Google Maps, whose API is used in almost every website that includes a description of how to get somewhere.

DEPLOYMENT MODEL AND TARGET AUDIENCE

The target audience is the line of business which leverages API Portals to think about new digital products to increase revenue or make customers happy. A key for success in a hybrid integration architecture is a good cooperation between API Management and different integration solutions. This enables developers to reuse services to concentrate on new features, shorter time-to-market, and innovation instead of recreating existing services again.

SOME OPTIONS?

Examples for API Management solutions are Apigee (Pure Player), Akana (Pure Player, formerly SOA Software), Mashery (TIBCO), or 3scale (Red Hat). Sometimes a hardware API Gateway such as IBM DataPower Gateway is used as replacement or in addition to the software API Gateway.

THE NEED FOR A HYBRID INTEGRATION ARCHITECTURE

This survey shows that a **single integration platform is not sufficient anymore** in the era of cloud, mobile, big data, and IoT. Differentiation between core services for running the business and edge services for changing the business is a key step towards a **Hybrid Integration Platform**. On-premise vs. cloud-native development and deployment is another key aspect. Streaming Analytics and Business Process Management (out of the scope of this article) are not part of application integration directly, but also relevant for a hybrid integration architecture.

KAI WÄHNER works as Technology Evangelist at TIBCO. Kai's main area of expertise lies within the fields of Big Data, Analytics, Machine Learning, Integration, SOA, Microservices, BPM, Cloud, Java EE and Enterprise Architecture Management. He is regular speaker at international IT conferences such as JavaOne, ApacheCon or OOP. References (presentations, articles, blog posts): www.kai-waehner.de, contact via @KaiWaehner or kontakt@kai-waehner.de.

Diving Deeper

INTO INTEGRATION

TOP #INTEGRATION TWITTER FEEDS TO FOLLOW RIGHT AWAY

 @simonbrown

 @hadleybeeman

 @olivergierke

 @samnewman

 @apireport

 @danielbryantuk

 @crichardson

 @davsclaus

 @discoposse

 @launchany

INTEGRATION ZONES LEARN MORE & ENGAGE YOUR PEERS IN OUR INTEGRATION-RELATED TOPIC PORTALS

Integration

dzone.com/integration

Enterprise Integration is a huge problem space for developers, and with so many different technologies to choose from, finding the most elegant solution can be tricky. 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.

IoT

dzone.com/iot

The Internet of Things (IoT) Zone features all aspects of this multifaceted technology movement. Here you'll find information related to IoT, including Machine to Machine (M2M), real-time data, fog computing, haptics, open distributed computing, and other hot topics. The IoT Zone goes beyond home automation to include wearables, business- oriented technology, and more.

Mobile

dzone.com/mobile

The Mobile Zone features the most current content for mobile developers. Here you'll find expert opinions on the latest mobile platforms, including Android, iOS, and Windows Phone. You can find in-depth code tutorials, editorials spotlighting the latest development trends, and insights on upcoming OS releases. The Mobile Zone delivers unparalleled information to developers using any framework or platform.

TOP EI REFCARDZ

Getting Started With Microservices
dzone.com/refcardz/getting-started-with-microservices
Learn why microservices are becoming the cornerstone of modern architecture, how to begin refactoring your monolithic application, and some common patterns that can help you get started.

Getting Started With JBoss EAP 7
dzone.com/refcardz/getting-started-jboss
Includes suggested configurations and extensive code snippets to get your Java application up and running inside a Docker-deployed Linux container.

Foundations of RESTful Architecture
dzone.com/refcardz/rest-foundations-restful
Brush up on REST architectural style, a worldview that can elicit desirable properties from the systems we deploy.

TOP EI WEBSITES

Largest API Directory on the Web
programmableweb.com/apis/directory

Enterprise Integration Patterns
enterpriseintegrationpatterns.com

Microservices Advice
microservices.com

TOP EI TUTORIALS

Microservices Architecture: What, When, and How
dzone.com/articles/microservices-architecture-what-when-how

5 Basic Rest API Guidelines
dzone.com/articles/5-basic-rest-api-design-guidelines

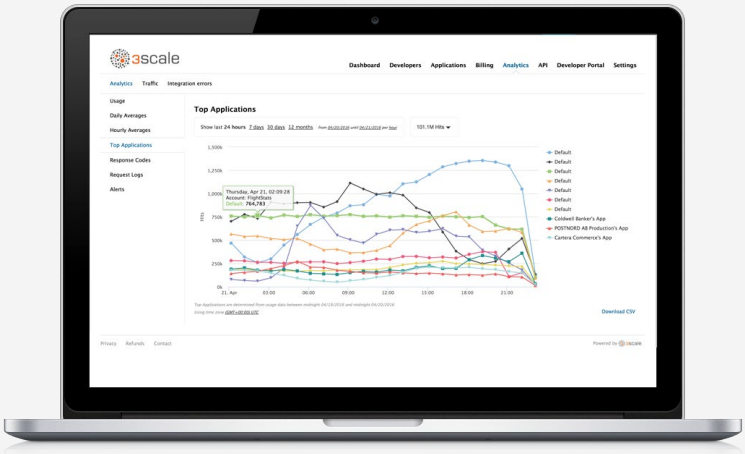
The Basics of Using Rest vs. SOAP
dzone.com/articles/rest-or-soap-which-one-to-use

3scale BY RED HAT

Future-proof flexbility and scale

3scale is the API infrastructure to build on now, and for the future.

Share, secure, distribute, control, and monetize your APIs using the platform built with customer control, performance, growth, and time-to-value in mind. Learn more at 3scale.net/dzone.



- Better time-to-value
- API traffic control
- API program tools

White paper:

Achieving Enterprise Agility with Microservices and API Management

Read now ›

APIs and Microservices Transform Infrastructure Hand in Hand

Microservices are proving to be a powerful architecture to break apart monolithic systems. Each new microservice created represents a separate bundle of functionality that can be independently updated and scaled. To get real value out of microservices, however, they need to be accessible and reusable throughout the organization. This is where APIs come in.

By associating a well defined API with each microservice, these components become first-class citizens within an infrastructure. Application development teams across the whole organization suddenly have a well defined, rich set of functionality to build on. With security, access control, and documentation, some of these microservices can also easily be made accessible to external players such as customers and partners.

Here are some recommendations:

- Ensure there are well defined interfaces for the functionalities provided by each microservice using a definition language such as the Open API Specification (openapis.org).
- Manage these services as products in their right: determine who uses it, how it should scale, and who would be affected by change.
- Add analytics and tracking for each API to determine usage patterns and help predict when delivery should be scaled.
- Add access control in the form of keys, oAuth, or other schemes to control who accesses services, and add rate limits and security for those microservices accessed outside your own team or company.
- Use container-based infrastructure to manage microservices deployment and provide the means to announce which microservices are available in a given environment, and enable discovery by potential users.

While the step from microservices to APIs may seem small, the added value is tremendous. Ensuring stability and security in interfaces, announcing roadmaps, tracking usage, and the other items make your microservices valuable well beyond the existing team.

WRITTEN BY STEVEN WILLMOTT
SENIOR DIRECTOR AND HEAD OF API INFRASTRUCTURE, **RED HAT**

PARTNER SPOTLIGHT

3scale API Management Platform By Red Hat

3scale's unique hybrid architecture creates flexibility, performance, and scale not achievable or cost-effective with other solutions.

CATEGORY
API management and integration

NEW RELEASES
Quarterly

OPEN SOURCE
Coming in 2017

CASE STUDY

In order to transition from a crowd-sourced idea to the most-used dataset of startup information, the CrunchBase team needed to build an API that functioned as a strategic business asset, in line with overall growth strategy. The team sought an API management solution that would both reduce operational costs and provide a flexible foundation for growth; ease of implementation and management were also key considerations. After a straightforward implementation of 3scale, CrunchBase saw a dramatic improvement in performance. Developer signups and API usage skyrocketed, and implementing 3scale led to decreased maintenance time, allowing CrunchBase's engineering team to spend more time working on improvements to the API itself.

STRENGTHS

- Flexible deployment options including an on-premise or cloud-hosted API gateway
- Single interface for control and visibility
- Scalable, high-performance architecture
- Built-in developer portal CMS and interactive documentation
- Highly-customizable security and access control features
- Now also works with Red Hat's strong middleware and microservices solution stack

NOTABLE CUSTOMERS

- Optimizely
- VictorOps
- Coldwell Banker
- Schiphol Airport
- UC Berkeley
- Transport for London
- SITA Labs
- Springer Verlag
- Campbell's Soup

BLOG 3scale.net/blog | **TWITTER** @3scale | **WEBSITE** 3scale.net

Executive Insights on Application and Data Integration

BY **TOM SMITH**
RESEARCH ANALYST AT **DZONE**

QUICK VIEW

01 APIs are the key to integrating applications and data with RESTful API's being the most frequently referenced.

02 The cloud, microservices, and the continued rapid expansion, collection, ingestion, and analysis of data in real-time are driving change with integration of application and data.

03 Open Source serves as the underpinning upon which most applications and data integration solutions are based.

To gather insights on the state of application and data integration, we spoke with 18 executives from 15 companies who are involved in the integration of applications and data.

Here's who we talked to:

SHAWN RYAN, VP Marketing Digital as a Service, Axway

KURT COLLINS, Director of Technology Evangelism and Partnership, Built.io

THOMAS HOOKER, VP of Marketing, CollabNet

PIYUSH MEHTA, CEO, Data Dynamics

DANIEL GRAVES, VP of Product Management, Delphix

SAMER FALLOUH, VP of Engineering, and

ANDREW TURNER, Senior Solutions Engineer, Dialexa

ANDREW LEIGH, VP of Marketing and Alliances, Jitterbit

TREVOR HELLEBUYCK, CTO, Metalogix

MIKE STOWE, Developer Relations Manager, MuleSoft

ZEEV AVIDAN, VP Product Management, Open Legacy

SEAN BOWEN, CEO, **GORDON MCKINNEY,** Senior Solution Architect, and

ROSS GARRETT, Product Marketing, Push Technology

JOAN WRABETZ, CTO, Quali

RAZI SHARIR, VP of Products, Robin Systems

GIRISH PANCHA, CEO, StreamSets

BOB BRODIE, CTO, SUMOHeavy

KEY FINDINGS

01 **The keys to integrating applications and data are APIs, standards, knowing business objectives, providing an excellent user experience, and testing securely.** It's all about RESTful APIs. APIs are the de facto standard, but one size does not fit all. Leverage standards and ensure you are using one data format to connect everything. Have a clear plan based on the objectives of the business and ensure all key players are in alignment. Everything needs to be customized for the client to ensure they receive a good digital, connected experience for customers, partners, and employees.

02 Cloud, microservices, and more data on more platforms are the most significant drivers affecting change in the integration of applications and data. The proliferation of publicly consumable cloud services with REST and JSON enable you to quickly bring services to market and therefore bring more value to customers more quickly. API middleware is replacing SOAP with RESTful APIs. There's a proliferation of applications, databases, and interfaces as the amount of unstructured data doubles every 18 months due to IoT and the propensity to hold data in perpetuity.

03 The technical solution most frequently used to integrate applications and data is **Open Source, though others were mentioned as well.** They use Open Source capabilities that are high in quality and security. Others build their solution on top of JSON and REST for an interface that allows customers to build an integration when, where, and how they want. Although some people claim to have completely proprietary solutions, others believe no one in this day and age is 100% proprietary as the market moves too fast to rely on a single solution.

04 **Real world problems being solved** by the integration of applications and data are broad with clients demanding real-time ingestion and analysis to provide real-time solutions including:

- Insurance companies enabling field agents to use mobile devices to issue quotes, see leads, and create scenarios for their clients.
- Reducing the friction of money transfers for mobile banking customers.
- Macadamian working with Phillips Connected Hue Lights to build a connected meeting room that lets people know when the room is occupied or available, while managing schedules and calendars.
- A major TV network using real-time KPIs to monitor viewers per show, ad revenue, and monitoring glitches to mitigate revenue loses.
- A portable defibrillator company using their platform to update firmware remotely with traceability to ensure all machines have up-to-date firmware.
- Vinli making non-smart cars smart.
- ParkHub creating a parking platform for large public events which optimizes revenue and reduces shrinkage.
- Robin creating a lawn care application that enables users to manage lawn maintenance from their mobile devices.
- Cisco providing search index optimization.
- Lithium doing product optimization for their community.
- A gaming company offering in-play (a.k.a. "real time") betting.
- An online mortgage service allowing consumers to be approved for a home mortgage in eight minutes.
- Club Auto in Canada providing an app that enables customers to request a tow truck and see the location of the tow truck as it's on the way.

Customers expect real-time data from their mobile devices without having to take any action, as well as an Apple-quality user experience.

05 There are several issues affecting the integration of application and data: **the scale and complexity of the data and the applications, service providers over-promising and under-delivering, and demands from inside the company to integrate faster**. Data integration is difficult. Companies must bridge the gap between legacy systems and open standards without losing data or risking its security. Customers buy apps and believe they will meet 100% of their requirements out of the box. A lot of companies claim to provide all the needed services with scale and security; however, as the application scales, latency becomes an issue and degrades the user experience. Businesses can't integrate applications and data quickly enough.

06 In the future, **the integration of data and applications will be more seamless, so "citizen developers" are able to get the information they want**. There will be continued growth of data, and the implications thereof will lead to the need for seamless integration. We will move to the citizen as programmer with the cost of entry dropping and the skillset necessary for development getting easier. There will continue to be explosions of endpoints and microservices connected at cloud speed supporting thousands of connections.

07 Skills that developers need in order to integrate applications and data include knowing how to **build APIs, integrate REST endpoints, and JSON**. Developers will need to understand why data is important, where it's going, and the systems it needs to integrate with. Know the types of data to use, how to use it, and how not to be killed by the scale and speed of data. Know the cloud, DevOps, continuous integration, and continuous development—that's where the future is. Stay abreast of what's new, particularly on the backend of the database. Do not let your knowledge become dated. Be able to see the other person's perspective.

08 The **majority of companies have, and enforce, API design and management policies.** While most companies are using API design and management policies internally, they are using their clients' standards for APIs.

09 **Virtually all companies are using microservices** themselves; however, they're seeing slow adoption by their clients. Nonetheless, they believe microservices will continue to grow in importance in the future.

10 **Additional considerations were wide ranging:**

- What are the challenges around data sovereignty in an open world?
- Integration is changing from a data exchange to integration with systems that perform like IoT with edge computing.
- Where are people seeing the hybrid cloud going? Will containers change cloud decisions in the modern world? Are people mixing and matching cloud, on-premise, and hybrid data?
- It's important to look at the expense of the solution relative to the value provided.
- Are people from different organizations structuring data entry for data flow and data in motion? Are there new roles and responsibilities? Centers of Excellence have been created in other layers of the data center, should we have this for data flow given the movement of data between nodes?
- Look at Open Source and proprietary management systems and decide what's right based on the culture and the nature of the enterprise.
- We aren't thinking enough about efficiency. We are generating and consuming more data every day, but the network is not growing. How do we keep up? Data is the key component of everything we do. Don't overlook the infrastructure to capture, store, manage, and analyze.
- The more we move towards a microservices approach, the more important the overall integration strategy to avoid architectural death by a thousand cuts.

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.

INTEGRATING THE SPRING CLOUD NETFLIX FRAMEWORK

Into Your Existing API

BY **JOHN VESTER**
DEVELOPMENT MANAGER AT **NEXTGEAR CAPITAL**

QUICK VIEW

01 As usage and dependency on your API continues to increase, analysis should take place to consider scalability and fault tolerance.

02 Netflix OSS Spring Cloud framework, build upon Sprint Boot, provides an excellent starting point toward enhancing the scalability of your API(s) without introducing a great deal of complexity.

03 Hystrix and Turbine can introduce circuit-breaker functionality into your environment, to handle scenarios where service calls become unresponsive.

By this point, your RESTful API development has likely reached a level of success within your organization. As more and more microservices or APIs emerge into your landscape, the dependency on your APIs likely grows as well. An increase in demand for your service correlates to a rising desire to make your API or microservice as robust as possible. Often, this approach translates to multiple instances running, with some form of load balancing put into place to keep up with demand.

NETFLIX (OSS) SPRING CLOUD PROJECT

With the popularity and success of Netflix's (OSS) Spring Cloud project, it might be time to consider integrating the framework into your mission-critical APIs. The Netflix project builds upon the Spring Boot framework and introduces the following components:

- Eureka is used for service discovery of instances running Spring-managed beans.
- Zuul handles routing services and is viewed as the gatekeeper for requests.
- Ribbon is used for dynamic routing and load balancing
- Hystrix offers circuit-breaker functionality to handle unresponsive API calls.
- Turbine provides Hystrix with information regarding all available circuit-breakers.

The use of Spring Cloud Config is recommended as well, as it allows for application configurations to be centralized in a Git-based repository.

FIRST CHALLENGE - GETTING TO SPRING BOOT

If your existing API or microservice is built on Spring Boot, you are in a very good position and can proceed to the next section. Most likely, your service is not running on Spring Boot, which means some effort will need to take place to get you in position to where you can take advantage of the OSS tools Netflix has offered the software community.

For starters, review the Convert an existing project to Spring Boot section in the current Spring Boot reference guide. The section provides some considerations and suggestions on how to perform the conversion. The section notes that non-web applications, like API services, are less difficult in making the transition to becoming a Spring Boot application. In this scenario, the code that creates an ApplicationContext is replaced with calls to SpringApplication. In that instance, the SpringBootServletInitializer can be extended for the Application class and Spring Boot auto configuration enabled:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application extends
SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
    SpringApplicationBuilder application) {
        return application;
    }
}
```

The application can be made executable by adding the following `main()` method:

```
public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
}
```

For existing Servlet applications, if they are utilizing version 3.0+ applications, the translation should migrate pretty easily—especially if using Spring Servlet initializer support classes.

For more complex applications, analysis should be completed to determine if the best approach is to start with a fresh Spring Boot application instance and migrate your classes and methods into the application.

Once you have your Spring Boot application running and APIs validated, the Netflix components can be introduced.

CREATE THE EUREKA AND ZUUL SERVERS

The first server that will be added will be the Eureka server, which will handle the service discovery. The Eureka server is a standard Spring Boot application with a simple `main()` method:

```
@SpringBootApplication
@EnableEurekaServer
@EnableDiscoveryClient
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(
            EurekaApplication.class, args);
    }
}
```

The Zuul server will act as the gatekeeper, or the primary server ultimately being contacted for your microservice. Here, another standard Spring Boot application instance is created with the following `main()` method:

```
@SpringBootApplication
@Controller
@EnableZuulProxy
public class ZuulApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(
            ZuulApplication.class)
            .web(true).run(args);
    }
}
```

At this point the Zuul server is up, running, and ready to handle the requests that are registered with Eureka. By default, the Ribbon service will be running as well, which will act as a load balancer for the clients accessing the services through Zuul.

UPDATING YOUR MICROSERVICE/API

In order for your service to register with Eureka, the `@EnableDiscoveryClient` annotation will need to be added to your Application class, as shown in this simple example:

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

When your service starts, it will register with Eureka, which can then be accessed via the Zuul gateway server. At this point, you will be able to introduce multiple instances of your service, which will be automatically registered and load balanced via Ribbon, which is running as part of the Zuul server.

ADDING HYSTRIX AND TURBINE INTO THE MIX

At this point, clients can access your service through the Zuul service. From a monitoring perspective, Hystrix can be added just as easily as the Eureka and Zuul servers were introduced—by starting with a base Spring Boot image and using the `@EnableHystrixDashboard` annotation.

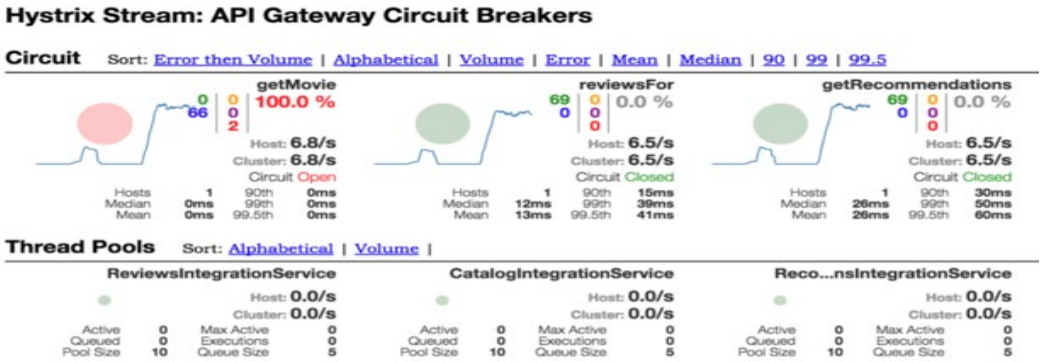
```
@SpringBootApplication
@Controller
@EnableHystrixDashboard
public class HystrixDashboardApplication extends
SpringBootServletInitializer {
    public static void main(String[] args) {
        new SpringgApplicationBuilder(
            HystrixDashboardApplication.class)
            .web(true).run(args);
    }
}
```

To set up Turbine, add the `@EnableTurbineAmqp` annotation to another base Spring Boot image:

```
@SpringBootApplication
@EnableTurbineAmqp
@EnableDiscoveryClient
public class TurbineApplication {
    public static void main(String[] args) {
        new SpringApplicationBuilder(
            TurbineApplication.class).run(args);
    }
}
```

ADDING CIRCUIT-BREAKER FUNCTIONALITY

The Netflix Hystrix service provides circuit-breaker functionality to your service consumer. If the server stops responding, Hystrix can redirect the API call to an internal method in the service itself. This way, your application can handle the scenario when a service is not responding.



Hystrix circuit-breaker opens and allows for a fallback method to be utilized.

The power and functionality of the Netflix's Spring Cloud project is quite impressive, but it is not without some challenges.

Hystrix has the ability to open the circuit and "fast fail" (bypass the service call and simply use the internal method) on every future call until the service becomes available again.

The Hystrix product provides a dashboard, which provides a dynamic list of the services being monitored.

In the image above from the Hystrix dashboard, the getMovie API call is currently not responding, causing the circuit to remain in an Open state, which would trigger calling any configured fallback methods.

To add the circuit-breaker functionality to your application, the `@EnableCircuitBreaker` annotation is added to the Spring Boot application for your service. From there, the `@HystrixCommand()` annotation can be used, similar to the example below:

```
@HystrixCommand(fallbackMethod = "baseResults")
public ResponseEntity<List<Result>> getResults(Long id) {
    ...
}

public ResponseEntity<List<Result>> getBaseResults(Long
id) {
    ...
}
```

In the example above, when the service call performed in the `getResults()` method encounters a service/time-out error, the `getBaseResults()` method will be used until the circuit is closed—which happens once the failed service issue is back online.

CHALLENGES

By this point, your core application/API is running as a Spring Boot application with one or more instances. Those instances are registered with the Eureka service and are being served by requests to the Zuul gateway, which includes Ribbon for load balancing. When a service stops responding, due to network/service timeouts, the

The biggest challenge is the impact on the developer, who has likely become accustomed to working with a single application service instance running. Now, the developer has to make sure at least two additional services (Eureka and Zuul) are running—which adds to the complexity of doing local development work.

Of course, things can get complicated when depending on another service also served by Zuul and Eureka—which may not have the option of running locally. In this case, the developer may need to register the local server instance with Eureka under another name—which can present challenges when used with a centralized configuration service.

LOOKING AHEAD

This article is intended to be a high-level review of the components involved with the Netflix's Spring Cloud project. Discussion around multiple Zuul and Eureka instances were omitted intentionally, but would be considerations for Production environments. Additionally, I didn't get deep into the details of Spring Configuration, the use of RabbitMQ, and more functionality of Hystrix/Turbine for the very same reasons.

If your API or microservice has reached the point to where it is a strong candidate for the features and functionality that Netflix's Spring Cloud project provide, I strongly recommend taking time to evaluate the usage of OSS into your solution. Even if the project requires time/effort to convert your service to Spring Boot, the long-term gains should outweigh the short-term costs.

JOHN VESTER is an Information Technology professional with 25+ years of expertise in application development, project management, system administration, and team supervision. He is currently managing an architecture/application development team based upon object-oriented programming languages and frameworks; has prior expertise building Java-based APIs against React and Angular client frameworks; and prior experience using both C# (.NET Framework) and J2EE (including Spring MVC, JBoss Seam, Struts Tiles, JBoss Hibernate, Spring JDBC).

Solutions Directory

This directory includes integration tools for API management, integration platforms, message queues, frameworks, and SOA governance, letting you find the solution that's right for your integration needs. Solutions are selected for inclusion in the directory based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

PRODUCT	COMPANY	TYPE	OPEN SOURCE	WEBSITE
3Scale API Management	Red Hat	API Management	No	3scale.net/api-management
Actian DataCloud	Actian	Integration Platform/PaaS	No	actian.com/products/data-integration/#DataCloud
ActiveMQ	Apache Software Foundation	Message Queue	Yes	activemq.apache.org/download.html
Adaptris Interlok	Adaptris	Integration Platform/PaaS	No	adaptris.com/pages/products-and-services/interlok
Adeptia Integration Suite	Adeptia	Integration Platform/PaaS	No	adeptia.com/products/adeptia-integration-suite
Akana API Management	Akana	API Management	No	akana.com/solution/api-management
Akana Software Suite	Akana	SOA Governance	No	akana.com/solutions/integrated_soa_governance
Amazon SQS	Amazon	Message Queue	No	aws.amazon.com/sqs
Anypoint Platform by Mulesoft	MuleSoft	API Management	No	mulesoft.com/platform/enterprise-integration
Apache Camel	Apache Software Foundation	Integration Framework	Yes	camel.apache.org/index.html
API Manager Platform by Cloud Elements	Cloud Elements	API Management	No	cloud-elements.com/platform
Apigee Edge	Apigee	API Management	No	docs.apigee.com/api-services/content/what-apigee-edge
apiman	Red Hat	API Management	Yes	apiman.io/latest
Artix by Micro Focus	Micro Focus	ESB	No	microfocus.com/products/corba/artix#
AtomSphere by Dell Boomi	Dell	Integration Platform/PaaS	No	boomi.com/integration/integration-technology
Aurea Sonic ESB	ESW Capital Group	ESB	Yes	aurea.com/legal/sonic-esb
Axway API Management	Axway	API Management	No	axway.com/en/enterprise-solutions/api-management-solutions

PRODUCT	COMPANY	TYPE	OPEN SOURCE	WEBSITE
Azuqua	Azuqua	Integration Platform/PaaS	No	azuqua.com
BizTalk Server by Microsoft	Microsoft	Integration Platform/PaaS	No	microsoft.com/en-us/cloud-platform/biztalk
Built.io Flow	Built.io	Integration Platform/PaaS	No	built.io/products/flow
CA API Management	CA Technologies	API Management	No	ca.com/us/products/api-management.html
Cazoomi	Cazoomi	API Management	Yes	cazoomi.com
Celigo	integrator.io	Integration Platform/PaaS	No	celigo.com/ipaas-integration-platform
CentraSite by SoftwareAG	Software AG	SOA Governance	No	softwareag.com/corporate/products/aris_alfabet/ea/products/centrasite/capabilities/registry_repository.asp
Cleo Integration Suite	Cleo	ESB	No	cleo.com/products/cleo-integration-suite
Cloak Labs	Cloak Labs	Integration Platform/PaaS	No	cloaklabs.com
CloudHub by Mulesoft	MuleSoft	Integration Platform/PaaS	No	mulesoft.com/platform/saas/cloudhub-ipaas-cloud-based-integration
CX Messenger by Aurea	ESW Capital Group	ESB	No	aurea.com/customer-experience-platform/cx-messenger
DreamFactory	DreamFactory Software Inc.	API Management	No	dreamfactory.com
E2E Bridge	Scheer	Integration Platform/PaaS	No	e2ebridge.com/en
Elastic Integration Platform by SnapLogic	SnapLogic, Inc.	Integration Platform/PaaS	No	snaplogic.com/solutions
Ensemble by Intersystems	InterSystems Corporation	ESB	No	intersystems.com/our-products/ensemble/ensemble-overview
evolution by Mertech	Mertech Data Systems, Inc.	API Management	No	mertech.com/evolution
Fabric8	Red Hat	Integration Platform/PaaS	Yes	fabric8.io
Fanout Zurl	Fanout, Inc.	API Management	Yes	fanout.io/download
Fiorano API Management	Firoano Software and Affiliates	API Management	No	fiorano.com/products/api-management.php
Fiorano ESB	Firoano Software and Affiliates	ESB	No	fiorano.com/products/open-source-esb/fiorano-esb
Fiorano Integration	Firoano Software and Affiliates	SOA Governance	No	fiorano.com/products/fiorano-integration-platform
FioranoMQ	Firoano Software and Affiliates	Message Queue	No	fiorano.com/products/Enterprise-Messaging/JMS/Java-Message-Service/FioranoMQ.php
Flowgear	Flowgear	Integration Platform/PaaS	No	flowgear.net

PRODUCT	COMPANY	TYPE	OPEN SOURCE	WEBSITE
Fujitsu Business Operations Platform	Fujitsu	Integration Platform/PaaS	No	fujitsu.com/global/products/software/middleware/application-infrastructure/interstage/solutions/bop
GXS Enterprise Gateway by OpenText	OpenText Corp.	ESB	No	opentext.com/what-we-do/products/business-network/b2b-integration-services
HANA Cloud Integration by SAP	SAP	Integration Platform/PaaS	No	help.sap.com/cloudintegration
HornetQ	Red Hat	Message Queue	Yes	hornetq.jboss.org
HP SOA Systinet	Hewlett Packard Enterprise Development LP	SOA Governance	No	hp.com/us/en/software-solutions/api-management-soa-governance
IBM API Management	IBM	API Management	No	ibm.com/software/products/en/api-connect
IBM Integration Bus	IBM	ESB	No	ibm.com/software/products/en/integration-bus-advanced
IBM MQ Advanced	IBM	Message Queue	No	ibm.com/software/products/en/mq-advanced
IBM Tivoli	IBM	SOA Governance	No	ibm.com/software/in/tivoli
ILANTUS Xpress Governance	ILANTUS Technologies Pvt. Ltd	SOA Governance	No	ilantus.com/xpress-governance
Informatica iPaaS	Informatica	Integration Platform/PaaS	No	informatica.com/products/integration-platform-as-a-service.html
InterSystems SOA	InterSystems Corporation	SOA Governance	No	intersystems.com/our-products/ensemble/enterprise-service-bus-esb-service-orientated-architecture-soa/ensembles-service-oriented-architecture-soa-capabilities/
IronMQ	Iron.io	Message Queue	No	iron.io/platform/ironmq
JaxView by Managed Methods	Managed Methods	SOA Governance	No	managedmethods.com
JBoss Fuse	Red Hat	ESB	Yes	redhat.com/en/technologies/jboss-middleware/fuse
Jitterbit	Jitterbit	Integration Platform/PaaS	No	jitterbit.com
JNBridge	JNBridge LLC	Integration Framework	No	jnbridge.com
K3 Integration Platform	BroadPeak	Integration Platform/PaaS	No	broadpeakpartners.com/data-integration-platform
Kapow Data Integration Platform by Kofax	Kofax Limited by Lexmark	Integration Platform/PaaS	No	kofax.com/products/business-intelligence-and-analytics/kofax-analytics-for-kapow
LegaSuite Integration	Rocket Software, Inc.	Integration Platform/PaaS	No	rocketsoftware.com/products/rocket-legasuite/rocket-api
Liaison Alloy	Liaison Technologies	Integration Platform/PaaS	No	liaison.com/liaison-alloy-platform
Microsoft Azure API Management	Microsoft	API Management	No	azure.microsoft.com/en-us/services/api-management

PRODUCT	COMPANY	TYPE	OPEN SOURCE	WEBSITE
MID Innovator	MID GmbH	SOA Governance	No	mid.de/en/business-activities/tools/innovator
Mule ESB	MuleSoft	ESB	Yes	mulesoft.com/platform/soa/mule-esb-open-source-esb
nanoscale.io	nanoscale.io	Integration Platform/PaaS	No	nanoscale.io
Neuron ESB	Neuron ESB	ESB	No	neuronesb.com
NServiceBus by Particular Software	Particular Software	ESB	Yes	particular.net/nservicebus
Oracle Service Bus	Oracle	ESB	Yes	oracle.com/technetwork/middleware/service-bus/overview
Oracle SOA Suite	Oracle	SOA Governance	No	oracle.com/us/products/middleware/soa/suite/overview
Point.io	Point.io LLC	API Management	No	point.io
Pokitdot	PokitDot, Inc.	API Management	No	pokitdok.com
ProSyst OSGi by Bosch Software Innovations	Bosch Software Innovations GmbH	API Management	No	prosyst.com/overview
RabbitMQ by Pivotal	Pivotal Software, Inc.	Message Queue	Yes	network.pivotal.io/products/pivotal-rabbitmq
Redis	RedisLabs	Message Queue	Yes	redis.io
Restlet Framework	Restlet, Inc.	API Management	Yes	restlet.com/projects/restlet-framework
Rocket Data	Rocket Software, Inc.	Integration Platform/PaaS	No	rocketsoftware.com/products/rocket-data
RunMyProcess by Fujitsu	Fujitsu	Integration Platform/PaaS	No	runmyprocess.com
SAIFE	SAIFE, Inc.	Message Queue	No	saifeinc.com
SAP API Management	SAP	API Management	No	go.sap.com/product/technology-platform/api-management.html
SAP HANA	SAP	Integration Platform/PaaS	No	hcp.sap.com
SAP NetWeaver	SAP	SOA Governance	No	go.sap.com/community/topic/netweaver.html
Seeburger	SEEBURGER	Integration Platform/PaaS	No	seeburger.eu/business-integration-suite.html
Sikka Software	Sikka Software Corporation	API Management	No	sikkasoft.com
SmarBear API Testing Solution	SmartBear Software	API Management	No	smartbear.com/solutions
Spring Integration by Pivotal	Pivotal Software, Inc.	Integration Framework	Yes	pivotal.io/agile/press-release/pivotal-releases-spring-framework-for-modern-java-application-development

PRODUCT	COMPANY	TYPE	OPEN SOURCE	WEBSITE
SQS by Amazon	Amazon	Message Queue	No	aws.amazon.com/sqs
SwarmESB	Unable to verify	ESB	Yes	github.com/salboaie/SwarmESB
Talend API Management	Talend	API Management	No	talend.com/blog/2015/02/09/defining-your-%E2%80%9Cone-click%E2%80%9D
Talend ESB	Talend	ESB	Yes	talend.com/products/application-integration
Talend Integration Cloud	Talend	Integration Platform/PaaS	No	talend.com/products/integration-cloud
Tibco	TIBCO Software Inc.	API Management	No	tibco.com
TIBCO ActiveMatrix	TIBCO Software Inc.	Integration Platform/PaaS	No	tibco.com/products/automation/application-integration/activematrix-businessworks
TIBCO ActiveMatrix	TIBCO Software Inc.	SOA Governance	No	tibco.com/products/automation/application-integration/activematrix-businessworks
TIBCO EMS	TIBCO Software Inc.	Message Queue	Yes	tibco.com/products/automation/enterprise-messaging/enterprise-message-service
TIBCO Hawk	TIBCO Software Inc.	SOA Governance	No	tibco.com/products/automation/monitoring-management/enterprise-monitoring/hawk
TIBCO Mashery	TIBCO Software Inc.	API Management	No	mashery.com/api-management/saas
Tyk.io On-Premise	Tyk.io	API Management	No	tyk.io/api-management-platform
UltraESB by AdroitLogic	AdroitLogic	ESB	Yes	adroitlogic.org/products/ultraesb
Vigience Overcast	Vigience	Integration Platform/PaaS	No	overcast-suite.com
WebMethods by SoftwareAG	Software AG	Integration Suite	No	softwareag.com/corporate/products/az/webmethods/default.asp
WebOTX ESB by NEC	NEC	ESB	Yes	jpn.nec.com/webotx/download/manual/92/serviceintegration/esb
WSO2 API Manager	WSO2	API Management	Yes	wso2.com/products/api-manager
WSO2 Carbon	WSO2	Integration Platform/PaaS	Yes	wso2.com/products/carbon
WSO2 ESB	WSO2	ESB	Yes	wso2.com/products/enterprise-service-bus
WSO2 Governance Registry	WSO2	SOA Governance	Yes	wso2.com/products/governance-registry
Zapier	Zapier	API Management	Yes	zapier.com
ZeroMQ	iMatix Corporation	Message Queue	Yes	zeromq.org

GLOSSARY

API MANAGEMENT PLATFORM Middleware used to oversee the process of publishing, promoting, and configuring APIs in a secure, scalable environment; platforms usually include tools for automation, documentation, versioning, and monitoring.

APPLICATION PROGRAMMING INTERFACE (API) A software interface that allows users to configure and interact with other programs, usually by calling from a list of functions.

ATOMIC TRANSACTIONS The smallest, irreducible series of steps needed to perform required actions. If those steps span multiple systems or sites, then they are distributed.

BUSINESS PROCESS MANAGEMENT (BPM) A workflow management strategy used to monitor business performance indicators such as revenue, ROI, overhead, and operational costs.

CIRCUIT BREAKER A design pattern that detects failures and cuts off affected services before the problem can cascade or otherwise cause more severe damage to connected services.

DARK LAUNCH A method of backend testing wherein a subset of user queries are routed through a process so that the system can attempt to resolve them, revealing pain points, bottlenecks, and other problems.

EDGE SERVICES Functions that are relevant to business users but aren't mission-critical or strategically vital for the enterprise.

ENTERPRISE APPLICATION INTEGRATION (EAI) A label for the tools, methods, and services used to integrate software applications and hardware systems across an enterprise.

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, and distributed computing.

ENTERPRISE INTEGRATION PATTERNS (EIP) A growing series of reusable architectural designs for software integration. Frameworks such as Apache Camel and 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.

FAILOVER A process by which processes that are behaving abnormally are moved to standby equipment to prevent loss of service.

GATEKEEPER The primary server ultimately being contacted for a microservice.

HORIZONTAL SCALING The process of adding compute capacity by adding or clustering machines.

IDEMPOTENCE The ability for a method to be called many times without different outcomes or side effects.

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

INTEGRATION PLATFORM AS A SERVICE (IPAAS) A set of cloud-based software tools that govern the interactions between cloud and on-premises applications, processes, services, and data.

LOAD BALANCING The process of distributing traffic across servers or resources to keep flow and performance optimal.

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

MIDDLEWARE A software layer between the application and 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.

POLLING The process of sampling data to collect information and determine state.

REPRESENTATIONAL STATE TRANSFER (REST) A set of principles describing distributed, stateless architectures that use web protocols and 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.

TRY-CANCEL/CONFIRM (TCC) A simple, stateful architecture that puts transactions into a reserved state (try), then either reverts back to their initial state on a failure (cancel), or moves them to a final state (confirm).

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

INTEGRATION RESOURCES

Available on DZone.com:

REFCARD: GETTING STARTED WITH

MICROSERVICES

dzone.com/refcardz/getting-started-with-microservices

REFCARD:

CAMEL ESSENTIAL COMPONENTS

dzone.com/refcardz/essential-camel-components

REFCARD:

JAVA EE 7

dzone.com/refcardz/java-enterprise-edition-7

REFCARD: GETTING STARTED WITH

JBOSS EAP7

dzone.com/refcardz/java-enterprise-edition-7

REFCARD: GETTING STARTED WITH

GIT

dzone.com/refcardz/getting-started-git

REFCARD

CORE JAVA

https://dzone.com/refcardz/core-java

VISIT DZONE.COM/REFCARDZ

For the latest in Integration news and free developer resources