API-Driven DevOps

Strategies for Continuous Deployment

AUTHORS

Bill Doerrfeld
Vassili van der Mersch
Kristopher Sandoval
Chris Wood

NORDIC APIS
nordicapis.com

API-Driven DevOps

Strategies for Continuous Deployment

Nordic APIs

Contents

Supported by Curity

Nordic APIs was founded by Curity CEO Travis Spencer and has continued to be supported by the company. Curity helps Nordic APIs organize two strategic annual events, the Austin API Summit in Texas and the Platform Summit in Stockholm.

Curity is a leading provider of API-driven identity management that simplifies complexity and secures digital services for large global enterprises. The Curity Identity Server is highly scalable, and handles the complexities of the leading identity standards, making them easier to use, customize, and deploy.

Through proven experience, IAM and API expertise, Curity builds innovative solutions that provide secure authentication across multiple digital services. Curity is trusted by large organizations in many highly regulated industries, including financial services, healthcare, telecom, retail, gaming, energy, and government services across many countries.

Check out Curity's library of learning resources on a variety of topics, like API Security, OAuth, and Financial-grade APIs.

Follow us on Twitter and LinkedIn, and find out more on curity.io.

Preface

There once was a time when software products were launched; physically shipped in a CD-ROM to a storefront, purchased, and then likely left to rust after the user's initial installation.

Nowadays, nearly all code is shipped over the web, meaning that **continuous** software updates are not only achievable, but expected, whether for mobile, browser, or desktop user experiences. Especially as the digital services we use embrace a subscription billing format, the process of continually delivering many fine-tuned iterations has become increasingly more strategic.

Thus, philosophies around this development style have proliferated the industry in the past decade. **DevOps** embodies this shift. The historical boundaries between development and operation teams have ebbed, and as continuous deployment becomes the norm, the tooling space has exploded to help startups and enterprise developers alike embrace more automation, and more efficient product cycles.

So, throughout early 2016 we admittedly followed some industry trends and wrote a lot on DevOps and relevant tooling. In this compendium we include curated lists of tools and analysis of specific areas like:

- continuous integration/deployment
- Docker containers
- Automated testing
- configuration management
- IoT continuous integration
- DevOps as a corporate role
- Automated code generation

- and more…

So please enjoy *API-Driven DevOps*, and let us know how we can improve. Be sure to join the Nordic APIs newsletter for updates, and follow us for news on upcoming events.

Thank you for reading!

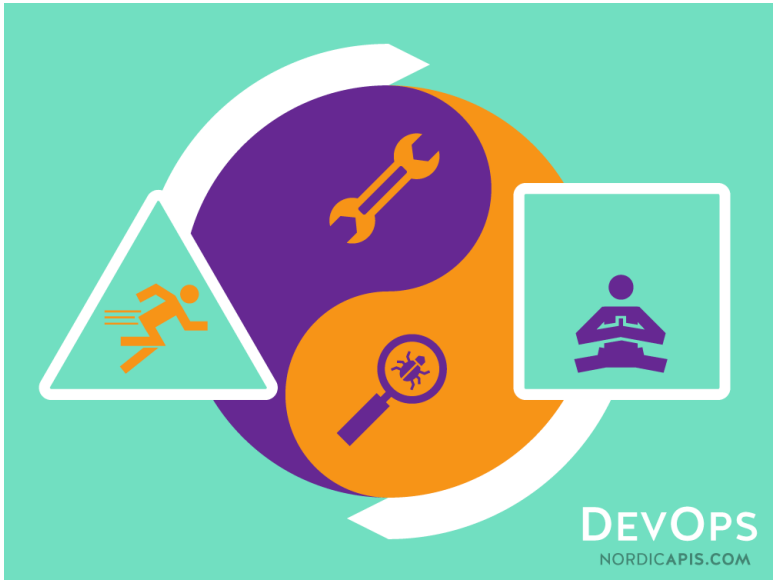
– Bill Doerrfeld, Editor in Chief, Nordic APIs

Connect with Nordic APIs:

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact

Defining the Emerging Role of DevOps



The nature of the IT industry is one of constantly emerging and refining processes and trends. One of these trends that has taken the IT world by storm is the emergence of DevOps. As more and more groups adopt DevOps organizational strata, understanding this new structure is key to keeping fresh and innovative.

A change to the very fundamental approach and organization of IT groups and API developers, **DevOps** is a firestorm that can empower and extend capability.

What is DevOps?

What exactly is DevOps? To understand what DevOps is, we first must take a step backwards. The development world is largely split into two groups - Development and Operations. Think of Development and Operations as the Felix and Oscar odd couples of the IT world — constantly fighting, at odds with one another, but ultimately dependent on one another.

Traditionally, Development is concerned with one thing — **development**. The continual innovation and experimentation resulting in iteration on previous versions is what drives the development of most applications and services. While this development process has created a wealth of services and functionality, it is at odds with the concept of "stability".

That's a problem, because that's what Operations is all about. Operations is concerned with **stability** above all else — the maintenance of systems and the stability of the forward facing application is the prime goal.

While many API providers can unify these groups, the fact is that the disconnect between purposes causes friction. This friction results in some huge issues for API developers, and drives much of the issues an API developer will face in the API lifecycle.

Why DevOps is Powerful

Enter DevOps. DevOps, a portmanteau of "Development" and "Operations", is a production philosophy that unifies the concept of experimentation and iteration with a measure of control to ensure stability. DevOps is incredibly powerful when implemented correctly.

The biggest source of increased power is the fact that implementing DevOps structuring makes an organization more agile. By tying in

iterative development with stability testing, organizations can surpass the time limitation often incurred between shipping versions between development teams and testing teams.

This joining is often referred to as "IT alignment", and its power simply can't be overstated. By properly aligning production and development, versions can be tested against the actual production environment, reducing the incidence of bugs and feature-set failures. Fundamentally, this reduction of segregated development procedures means greater results with less time and resources.

Implementing DevOps

The biggest change an API developer must make to support the switch from traditional segregated development to DevOps is a change to the fundamental development approach.

Development is commonly considered a process of "waterfall" interactions — one team does their work, shifts it down to another team, that teams does their work, and so on, culminating with the testing of the finished product and ultimate acceptance or denial.

While this is great on paper for ensuring compatibility and functionality, in actuality, all it does is put the development process through a corporate version of the "telephone game". Each transition from one team to another dulls the product, waters it down, and ultimately removes it one more layer from the original vision and purpose.

By changing the concept of development from "waterfall" to "point-to-point", developers can harness the power of collaborative work in a more functional way.

When a feature is proposed, DevOps should treat it as a journey from inception to creation, and test it throughout development. When a feature is completed and tested, it should be im-

plemented. Streamlining the process makes development more effective, quicker, and more efficient.

DevOps Tools

With increased power and responsibility, DevOps groups require more powerful and extensive tools to carry out their job. Thankfully, with the emergence of DevOps as a concept, DevOps tools have become more common. Throughout the course of this eBook we will outline many relevant tools, but take the following examples.

One tool for DevOps in production is the ELK Stack. Combining Elasticsearch, Logstash, and Kibana, the ELK Stack creates meaningful information from data culled from consumers. By examining usage demographics and behaviors, deploy issues can be identified, possible issues can be extrapolated pre-deploy, and behavioral trends and security issues can be determined.

Some tools, like Atlas, integrate DevOps as a process rather than a state of being. Atlas integrates deployment with automated workflows, constrained by set policies. By controlling the nature of deployment and tying it to the **development cycle**, Atlas allows for a seamless transition from segregated development and operations to the fabled DevOps.

A similar tool, Chef, utilizes "recipes" in place of policy-driven workflow automation to rapidly and automatically deploy services. Chef also has the added benefit of functioning in the open source realm, and boasts a collection of thousands of user-made "recipe" scripts, allowing for proven and rapid development regardless of the enterprise scale and scope.

Not all tools have to focus on automation, either. In an upcoming chapter we will discuss the power behind Docker, and this power is no more evident than with DevOps teams. Docker allows developer

to segregate services into separate containers, and allows for the quick "spinning up" of new environments for testing, development, and isolation. By allowing for mutli-modal production environments and isolating testing from deployment, deploys can be more thoroughly vetted throughout the streamlined process.

One of the biggest oversights in DevOps isn't necessarily the development cycle, however — it's the **development environment**. Even with powerful solutions like Docker, environments in production and in development can differ wildly. This often results in developers making changes to the base code that worked in development to work in production. For developers moving into DevOps, the environment must be unified.

Thankfully, there are just as many tools for exactly this purpose. One of these is ScriptRock GuardRail, a powerful monitoring and configuration system that checks the states of QA, production, and development environments. By ensuring that these environments are the same, DevOps can unify their functionality, and make their development that much more efficient.

For managing these production resources, there's yet another powerful tool — Stackify. Stackify aggregates errors, tests database queries and requests for speed and accuracy, and collates all of this into easy to read metrics. This is a great tool for ensuring that your production environment is as good as it can be, but is also great for stamping out issues in QA environments as they arise. Having an integrated solution such as this goes a long way towards making the adoption of DevOps possible and painless.

DevOps as a Career

DevOps is a unique thing to consider in that it functions as both a conceptual approach to development and as a specific talent. This makes it a strange thing to consider, as the expectations of strategy

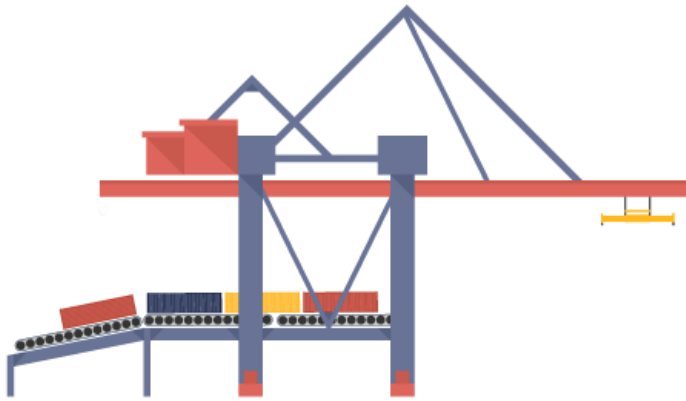
is one of a suite of approaches or concepts, and the expectations of a skillset is a single or unified group of skills — DevOps is the middle ground, a group of strategies as an organizational approach unified as a singular skillset, the joining of disparate approaches.

In other words, the combining of the "operations" skillset with the "development" skillset is, in itself, a skill, as is thus a marketable career. Accordingly, in the coming years, many professionals will begin seeing the job title shift from "development management with operational skillset" or "operations lead with development experience" into "DevOps manager".

Right now, however, this is somewhat nebulous, largely due to the fact that there is no "unified approach" in the landscape. While many organizations have adopted DevOps as a way of life with little issue, others have rebuked the concept as unnecessary at best, and bloat at worst. Some employers might find issue with oneself branding themselves as a "DevOps leader", while others would find issue NOT branding oneself as such.

The landscape is changing. As more and more companies transition to the DevOps approach, these issues with be far less common, and DevOps will be adopted by many organizations. For now, however, DevOps is best considered a "strategy development", and less so a career title.

Analysis: Less Uptime Sacrificed for new Features



There's a lot to be said for DevOps. While there will undoubtedly be issues switching from more traditional development cycles to the DevOps world, the fact is that DevOps is incredibly powerful, and is quickly becoming standard for enterprise businesses.

A speaker at DevOps Days put it best by saying:

> In most organizations, Dev and Ops have misaligned goals. Dev is measured by the number of new features. Ops is measured by 100% uptime. Question: What's the best way to get 100% uptime? Answer: Don't introduce any new features or make any changes.

It is this chief issue that so many developers have experienced that makes DevOps such a raging firestorm. Adoption now could help in the long run, making your API and enterprise more agile, creative,

and innovative. Not adopting it? Well, it could either have no effect whatsoever, or leave you in the dust.

The choice is pretty clear.

10 Continuous Integration Tools to Spur API Development

Software development these days is about iterating fast, and releasing often. In the 10 years since Martin Fowler wrote his original paper, **continuous integration** has become a cornerstone of the software development process. It is a fundamental part of Agile development (being one of the core tenants of Extreme Programming) and has helped to spawn organizational change with the creation of the DevOps approach to application development.

Within the API community, continuous integration (CI) tools generally fall into two categories. The first wave of tooling saw the role of APIs in CI as being one of a building block, with the CI

servers available tending to reflect this. Solutions such as Jenkins, TeamCity, Travis CI and Bamboo provide or consume APIs to help drive the build process, allowing automated builds to invoke actions during the course of execution: this role persists and is still expanding.

The second category of tools help organizations implement CI for the APIs they are *creating*: These tools can perform test execution and metric collection associated with running a CI process, allowing organizations to introduce a greater degree of automation in their approach to testing APIs.

In this post we'll cover the latter of the two categories and look at some examples of the tools available that help spur agile API development. While it is possible to create tests for your API using no more than shell script and cURL, this takes time and effort that may not be available, so using tools built for the purpose of testing APIs can be a great way of accelerating your efforts. At the time of this writing there is a burgeoning number of both open source and commercial solutions available that offer a mixture of SaaS-based and on-premise solutions. Ideally these solutions implement some of the following characteristics:

- Easy integration with CI servers using either plugins or webhooks;
- Automatic comprehension of APIs through ingestion of description specifications such as Swagger, RAML and API Blueprint;
- Polyglot support for test creation, giving users choice of the language they use to construct their integration tests;

Please note that this isn't intended to be a product review, rather a list of "honorable mentions" for of the solutions in this space and their features.

Abao

Abao is a Node.js module that allows developers to test their RAML API description against a backend instance of their API: To execute the tests, a developer simply executes the `abao` command line tool, passing the RAML file and API endpoint as parameters. Abao also does not provide CI server integration of the box, but scripting its execution from a build job definition would not be difficult to achieve.

DHC

DHC is a tool offered by Restlet that allows developers to create a suite of unit and functional tests for their own or any third-party API and includes support for security, authentication and hypermedia. It also includes the ability to perform scenario-based testing, with each scenarios allowing multiple features of an API to be tested.

In terms of CI server integration, Restlet offers a plugin for Jenkins and also a Maven plugin that can be incorporated into other solutions, allow developers to incorporate their tests into the build pipeline.

Dredd, by Apiary

In 2015 Apiary introduced continuous integration features into their development platform, enabling developments to continuously and automatically test their APIs with test definitions generated from their API Blueprint description. This functionality is implemented with Dredd, a "*language agnostic command-line tool for validating API documentation written in API Blueprint*".

Developers use Dredd to execute their tests, with the results posted in the Apiary development console.

Obviously, using Apiary and/or Dredd for continuous integration necessitates the use of API Blueprint to describe the API. Apiary also does not provide CI server integration out of the box, but again, scripting the execution of Dredd from a build job definition would not be difficult to achieve.

APIMATIC

APIMATIC is a solution that allows developers to automatically generate SDKs from API descriptions, which can be imported from existing API Blueprint, Swagger and RAML API descriptions. APIMATIC provides the functionality to initiate the code generation process from CI, by providing an API that can be called from a build job definition, allowing the compiled SDK to keep step with the state of the API automatically.

Chakram

Chakram is an API testing framework built on Node.js and Mocha. It implements a behavior-driven development style of testing (although it does not use Gherkin-style test definitions). Chakram does not provide CI server integration of the box, but like the other examples scripting it's execution from a build job definition would not be difficult.

Frisby.js

Frisby.js is a Node.js module for testing APIs, allowing developers to create tests in JavaScript. Whilst Frisby does not have built-in

integration with CI servers, again it can be called from a script-based build job definition using its command line interface. A notable feature that Frisby implements is the ability to generate JUnit format test reports, compatible with several CI servers including Jenkins and Bamboo, making it easier for test results to be displayed and understood.

Postman

Postman is a well-known and popular API testing client, with support for importing API descriptions in RAML, Swagger and WADL formats. Developers can create tests using JavaScript that can be bundled into "collections" for executing automated test suites.

Postman is accompanied by its Newman command line client that can be incorporated into a script-based build job definition on most CI servers. Postman provides a walkthrough of implementing this with Jenkins on their blog, where they use the built-in scripting capability to call the client, passing the required Postman collection as a parameter.

Runscope

Runscope markets itself as an API monitoring and testing tool capable of being used throughout an organization's development lifecycle, with capabilities such automatic test generation from a Swagger description or Postman collection.

Runscope appears to be one of the most mature tools in this space with several features for CI. It can be easily plugged into a build via a webhook, triggered at any point to initiate API testing, the scope of which includes validation of the data returned in an API response. Runscope also provides detailed information

on how to configure the webhook, what they call the "Trigger URL" for a series of CI servers and platforms. They also provide functionality that helps users to analyze their results by providing integration to analytics platforms, allowing detailed digestion of test results. Finally, Runscope has developed a plugin for Jenkins that is available in the Jenkins plugin repository, and has provided a walkthrough of the implementation.

SmartBear TestComplete Platform

SmartBear Software have a long history in both SOAP and REST API testing and are probably most well known as the providers of the commercial version of SOAP UI. SmartBear has entered the CI market with their TestComplete platform, which provides developers with the means to create API tests in a variety of languages and integrate them with SOAP UI. Moreover, these tests can also be integrated with Jenkins using a plugin, which allows the tests to be included in the build pipeline for a given API. For CI servers other than Jenkins, SOAP UI comes with a command line tool that could be called using script from a build job definition in the manner we've already mentioned above.

Swagger Diff

Swagger Diff is a command line tool created by Civis Analytics to test backward compatibility between two versions of a Swagger description for a given API, the idea being to plug this into a CI server using a script-based build job definition as described above. On finding a breaking change the tool will return an error, providing details of the diff that highlights the source of the incompatibility.

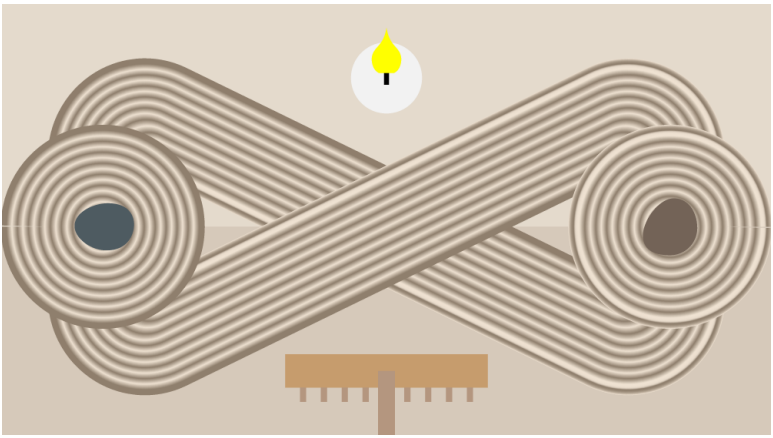
Analysis: CI is a Growing Presence in the API Economy

This compendium highlights the fact that continuous integration tools related to APIs are a growing presence in the API economy, with a mixture of both open source and commercial solutions available to organizations that can help them spur their API programs through effective, automated integration processes.

Another area we have not covered that is likely to be increasingly important is from the perspective of the API consumer; publicly available API descriptions, whether hosted at the API provider or in an API directory will become triggers for continuous integration, allowing build jobs to run based on published changes to the format, generating new client libraries for the API in question.

Real benefits for organizations lie in the practice of continuous integration. All API providers should consider leveraging the practice to ensure their efforts in exposing APIs have the best chance of success.

Reaching DevOps Zen: Tracking the Progress of Continuous Integration



Continuous Integration (CI) is a part of the **DevOps** process that aims at integrating code written by a team of developers. This involves building the code along with dependencies, testing it against a prepared set of unit and integration test suites, and creating reports listing the resulting artifacts.

By performing regular tests on the code in its entirety, spotting regressions early on, and producing stable artifacts to base future versions of the code on, CI helps avoid wasting time dealing with conflicts in the code that teams are collaborating on. This is why many consider it to be the cornerstone of **agile software development**.

CI is closely related to the idea of **Continuous Delivery**, in which

the successful builds that result from the continuous integration process are deployed to staging or even production environments at regular intervals. CI has become one of the mainstays of modern software engineering, and all serious development teams have set up one of the many available CI solutions to accelerate their common efforts.

Since building code and running tests are usually resource-intensive processes, teams were quick to eliminate on premise CI servers in favor of cloud-based SaaS products. Over the last few years, local building and testing of code has made way for **API-driven continuous integration**. As DevOps technologies have evolved of late, the software companies behind each of the major CI tools have all had to adapt to them as well.

Let's delve into what some of them have been doing, and then examine modern trends in continuous integration.

Traditional CI

Although CruiseControl was the first formal continuous integration tool, Jenkins was the first to gain widespread adoption. It was built in Java and open sourced in 2004 by a developer at Sun Microsystems named Kohsuke Kawaguchi (Jenkins is the largest fork of the Hudson project).

Using Jenkins, developers can set up a workflow to pull their code from a **version control repository** like Subversion, Git, or Mercurial, and trigger code builds, test runs, and deployments every time the repository's **mainline** has been updated.

Jenkins remains one of the most popular continuous integration tools on the market. While it was initially only meant to support Java, legions of developers have contributed plugins for many other languages.

For a modern development team, the main selling point in favor of Jenkins, beyond the option of an on-premise build server, is its **flexibility**. While its learning curve is somewhat longer than with hosted CI products, Jenkins can be customized to fit any DevOps workflow.

Initially Jenkins was only available as on-premise, but several cloud players have created **SaaS** solutions based on Jenkins. In particular, CloudBees, a company that previously offered PaaS hosting for Java and Scala applications, has recently hired Kawaguchi and pivoted to focus on their cloud-based Jenkins offering.

Jenkins is often compared to TeamCity by JetBrains, another traditionally on-premise Java-based CI tool, and Atlassian's Bamboo, which can be either on-premise or hosted on Amazon EC2 and features deep integration with other Atlassian products like JIRA, Bitbucket, and HipChat.

Test suites within continuous integration workflows have evolved over time. Initially only **unit tests** were included, testing individual objects and software components in isolation. While this was already much better than nothing, only running unit tests could leave out problems in the communication between components. Automated **integration testing** was then born to mitigate these risks, but it still wasn't enough to cover all bases. Indeed, even integration testing didn't cover the problems that could be encountered by the end user navigating a GUI or web page.

Functional testing, previously the exclusive domain of manual testers, has become fair game for automation. Teams started using tools like HTTPUnit and Selenium to record browsing behavior and then replay the tests every time a new build was created.

Continuous integration also enables teams to link software versions with features and bugs by including issue identifiers from bug tracking tools in build metadata. In this way, project managers are able to follow progress from the output of the CI workflow.

CI in the cloud

Fast forward to 2011, and many dev teams were tired of self-hosting their own continuous integration system, as configuring it often proved costly in time and resources — time that could be better spent working on applications. **SaaS** solutions quickly proliferated to fill this gap in the market.

Travis CI is a hosted continuous integration service built on top of the GitHub API. It lets dev teams build any project, provided that the code is hosted on GitHub. Travis reacts to triggers within a GitHub repository such as a commit or a pull request using **webhooks** to start tasks such as building a code base. It uses GitHub not only to fetch the code but to authenticate users and organizations.

Teams using cloud-based solutions like GitHub and Travis CI for **version control** and CI no longer face the headaches of managing these tools themselves as was necessary only a few years ago. What's more, since all actions on both Travis CI and GitHub can be triggered via their respective APIs, this workflow can be entirely **API-driven**.

Another advantage of hosted CI solutions is that they can provide much broader **testing facilities**. Browser and OS testing used to be a tedious affair — workstations and staff had to be dedicated to ensure that bugs didn't appear within a certain environment. In contrast, a hosted solution can maintain a set of cloud-based servers with different configuration for this purpose. Travis allows testing on Linux, Windows and Mac environments. Travis CI supports a range of programming languages such as PHP, Ruby, Node.js, Scala, Go, C and Clojure. It now supports both public and private repositories, but it was traditionally associated with open source software — Travis CI's free version is itself an open source project hosted on GitHub.

Similar tools out there include:

- TestCafe is a specialist service that lets teams record functional tests and runs them in a variety of browsers.
- CircleCI is a popular hosted continuous integration service. One of its differentiating features is its focus on performance. It makes heavy use of parallel computation and splits test suites to run tests in parallel. CircleCI supports many startup-friendly programming languages and it supports testing against many modern SQL and NoSQL databases like PostgreSQL, MongoDB and Riak.
- Codeship is similar to Travis and CircleCI but supports both GitHub and BitBucket for both code and credentials. It does continuous integration and continuous delivery, supporting deployments to PaaS hosting platforms like Heroku and Google App Engine. It also integrates with internal communication tools like HipChat and Campfire. Last year Codeship introduced ParallelCI, its answer to CircleCI's parallel running of test suites.

Mobile CI

The advent of smartphones has led to a cambrian explosion in mobile application development. Mobile apps have specific requirements in terms of their build, test and deployment processes. Some of the major CI vendors have a mobile CI offering — like CircleCI, who purchased Distiller to provide support for iOS apps.

We've also seen the emergence of mobile CI specialists, such as:

- Hosted-CI, a hosted CI solution focused on iOS apps (as well as desktop applications for OS X) built on top of Jenkins.
- GreenHouse, which supports native iOS and Android code as well as cross-platform frameworks like Cordova and Ionic.
- Bitrise is a similar solution. In addition to native Android and iOS apps, Bitrise supports apps built with Xamarin.

Indeed, mobile applications have different requirements in terms of testing and distribution, and different mechanisms for dependency management (such as Cocoapods for Swift and Objective-C apps).

Mobile testing also requires different software from web application testing. Mobile testing frameworks like Espresso for Android and Appium automate away much of the inherent difficulties, but since mobile apps — like native desktop apps — run on the client side and outside of the well-defined context of a browser, that still leaves a large grey area that developers don't like to leave unattended.

Crashlytics, now part of Twitter Fabric, has mobile **crash reporting** capabilities in addition to its built-in CI support. These alerts help dev teams analyze and react to difficulties encountered by users on their mobile devices. HockeyApp also reports crashes for mobile and desktop apps and enables you to test against simulated mobile environments.

Since there are wide varieties of mobile clients, especially in the Android world, it is impractical to have all testing performed in a centralized manner. Instead, **beta testing** has been embraced in the mobile development community, with tools like TestFairy and TestFlight to distribute tentative new versions of apps to beta testers.

Distribution of mobile apps is a fundamentally different problem when compared with web applications. Instead of simply pushing all the code, including client-side components to a server environment like Heroku or Amazon Web Services, a mobile app must go through the cumbersome **approval process** of the relevant app stores. This slows down the continuous delivery process and unfortunately introduces manual steps in an otherwise fully automated process.

Fastlane, also now part of Twitter Fabric, aims to streamline the delivery of apps by automating most elements of the app approval workflow, like taking screenshots of the new app and dealing with certificates. Fastlane is sometimes used in combination with a CI

tool like Jenkins.

Future of CI

Continuous integration and continuous delivery are here to stay. Most development teams wouldn't consider tackling a significant software project without setting up a CI workflow.

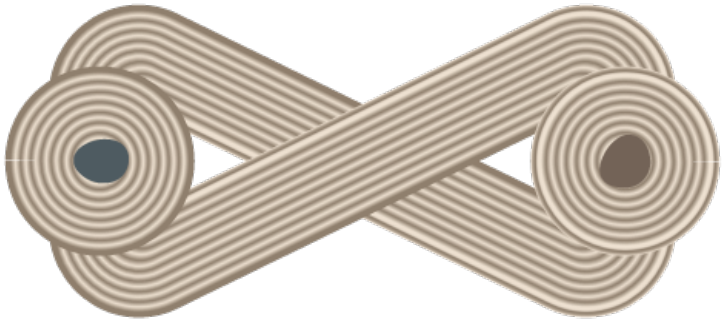
But it's far from a static discipline — there are constant innovations in the area of CI, and new fields of application. We've explained how CI is relevant to web and mobile development, but the coming years will show us how it will be integrated into development processes in smart watches, smart cars, and more generally throughout the internet of things. What will CI look like for virtual reality and biotech software?

APIs are increasingly becoming the bedrock of internet communications, and stability is paramount to a well-functioning API, so CI will increasingly matter in API development.

An ongoing problem in CI is the difficulty of automating tests in development environments that are not always similar to production. A recent evolution in DevOps is the growing popularity of Docker, a container technology that allows you to isolate applications and their dependencies from the underlying systems they run inside. Docker's portability has led several CI vendors to embrace Docker. CircleCI supports container-based applications and CodeShip recently introduced Jet, its solution to test and deploy Docker applications.

Docker simplifies modern CI; a throwaway build or test server is suddenly only an API call away. Docker also reduces the need for OS testing as it standardizes application runtime environments.

Analysis: CI is a Mainstay for Development



From its beginnings as an additional chore for developer teams and a somewhat reluctant concession to code quality, Continuous Integration has become one of the mainstays of modern software development. CI has has moved to the cloud and is gradually growing in scope; the next avenue for CI is perhaps the Internet of Things, an implication we are excited to explore soon. In the next upcoming chapters we'll track another important aspect of DevOps, namely Docker, so read on.

Introducing Docker Containers

One of the major issues universally faced in API development is the management, packaging, and distribution of dependencies. The dependency set required by an API might make it extensible, wonderful to use, and extremely powerful. However, if hard to manage, dependencies could spell adoption limbo.

A solution to this age-old problem has exploded onto the scene in recent years, however. Docker is a system by which a complete ecosystem can be contained, packaged, and shipped, integrating "code, runtime, system tools, system libraries - anything you can install on a server".

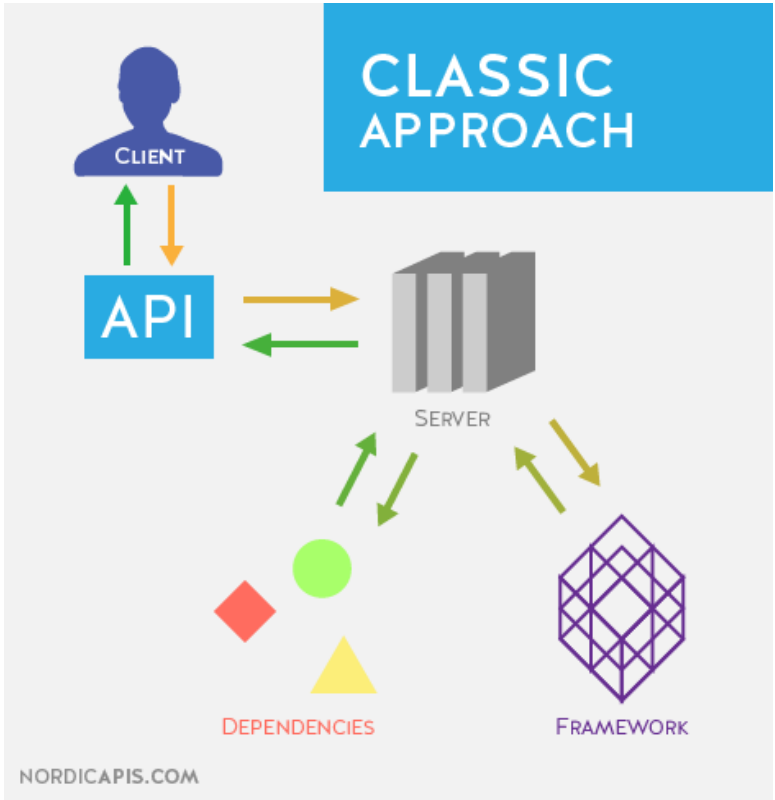
In this chapter we're going to take a look at Docker and its container system. We'll discuss some cases where using Docker is a good idea, some cases where it may not be the best solution, and the strengths and weaknesses of the system as a whole.

What is Docker?

Before we can critique it, we need to fully understand what Docker is, and how it functions. Simply put, Docker is a methodology by which the **entire development ecosystem** can be provided to API users and consumers in a singular application. Docker is a methodology to handle dependencies and simplify functionality, and can be used with a variety of Microservice languages.

The "Classic" API Approach

Consider the "classic" method of dependency handling and ecosystem management. An API is developed which issues remote calls to a server or service. These calls are handled by a framework that is referenced external to the API. This framework then requests resources external to the API server in the form of **dependencies**, which allow for the code to function in the methodology it was designed for. Finally, data is served to the client in the constrained format determined by the API.



Heavy and unwieldy, this system is **antiquated** in many ways. It depends on the developers of dependencies to update their systems, maintain effective versioning controls, and handle external security vulnerabilities.

Additionally, many of these dependencies are **proprietary**, or at the very least in the hands of a single developer. This means that code is maintained **external** to the API, and any change in functionality, failure in security, modification of additional dependencies used by the dependency author, and so forth can cause catastrophic failure.

Barring dependency issues, the "classic" approach is simply **resource heavy** and slow. It requires that developers host the entire

system in a web of interlacing APIs, attempting to hack together a system that functions. It's a delicate, functional ecosystem that is impressive in its complexity — but with this complexity comes the potential for the classic approach to become a veritable "house of cards".

The Docker Approach

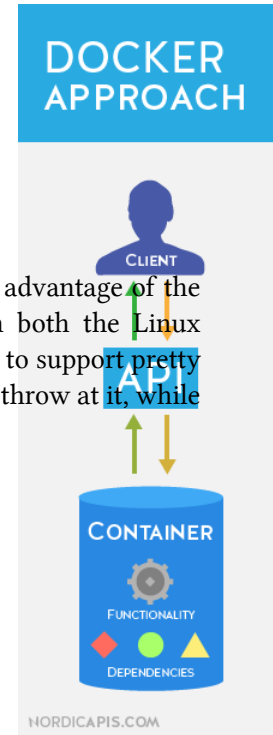
Docker has created a completely different approach. Instead of depending on multiple external sources for functionality, Docker allows for the remote use of operating system images and infrastructure in a way that distributes all the dependencies, system functionalities, and core services within the API itself.

Docker calls these "containers". Think of containers like virtual machines — but better.

A virtual machine (VM) packages an application with the binaries, libraries, dependencies, and an operating system — and all the bloat the comes with it. This is fine for remote desktop and enterprise workstation usage, but it leads to a ton of bandwidth waste, and isn't really a great approach for APIs.

Docker containers, on the other hand, are more **self sufficient**. They contain the application and all of its dependencies, but use a communal kernel with other applications in the userspace on the host operating system. This frees up the container to work on **any** system, and removes the entirety of the operating system bloat of virtual machines by restricting contents to only what the API or application needs.

Why Docker?



With its similarities to virtual machines, a lot of developers are likely wondering what the buzz about Docker is. What, specifically, makes it so great? There are a lot of reasons to love docker:

- **Open Source** - Docker is designed to take advantage of the wide range of open standards present on both the Linux and Microsoft OS ecosystem. This allows it to support pretty much any infrastructure configuration you throw at it, while allowing for transparency in the code base.

Unlike closed systems, open systems are routinely checked for security vulnerabilities by those who use them, and are thus considered by many to be "more secure". Additionally, because these standards are meant to promote interoperability between disparate systems, compatibility between systems in the code base or library functionality is non-existent.

- **Security Through Sandboxing** - Docker may not call it "sandboxing", but that's essentially what it is — every application is isolated from other applications due to the nature of Docker containers, meaning they each run in their own separate, but connected, ecosystem.

This results in a huge layer of security that cannot be ignored. In the classic approach, APIs are so interdependent

with one another that breaching one often results in the entire system becoming vulnerable unless complex security systems are implemented. With applications sandboxed, this is no longer an issue.

- **Faster Development and Easier Iteration** - Because a wide range of environments can be created, replicated, and augmented, APIs can be developed to work with a variety of systems that are otherwise not available to many developers.

As part of this benefit, APIs can be tested in enterprise environments, variable stacks, and even live environments before full deployment without significant cost. This process integrates wonderfully with Two-Speed IT development strategies, allowing for iteration and stability across a service. This has the additional benefit of supporting a truly effective microservices architecture style, due largely to the overall reduction of size and the lean-strategy focused methodology.

- **Lightweight Reduction of Redundancy** - By the very nature of Docker containers, APIs share a base kernel. This means less system resources dedicated to redundant dependencies and fewer instances to eat up server RAM and processing chunks.

Common filesystems and imaging only makes this a more attractive proposition, easing the burden of space created by multiple-dependency APIs by

orders of magnitude. This makes the API container simple to use and understand, making the API truly functional and useful.

Another representation of Docker architecture

Simple Docker Commands

Part of the appeal of Docker is how simple the commands and variables therein

are. For example, to create a container, you simply issue the following call:

```
POST /containers/create
```

Using a list of variables, you can completely change how this container functions, what its name is, and what the container has within. You can change everything from the name using:

```
--name=""
```

To the MAC Address of the container itself:

```
--mac-address=""
```

You can even change the way the container functions with the server itself, assigning the container to run on specific CPU cores by ID:

```
--cpuset-cpus=""
```

When the “docker create” command is run, a writable layer is created over the imaged kernel that allows modification of the entire application functionality.

These variables are also available for the “run” command. Most importantly, the “run” command can also attach a static address to the API container utilizing the “-p” and “–expose” calls:

```
docker run -p 192.168.0.1:8910
docker run --expose 8910
```

These two calls will first assign the container to port 8910 of the IP 192.168.0.1, and then expose that port to forward facing traffic (in effect opening the port completely for API functionality).

In order to make these containers functional, of course, a container needs to connect to an image. These images are built utilizing the "build" call:

```
docker build -t ubuntu
```

This builds a simple image, which is then given an "IMAGE ID" variable that can be called using the "docker images" call:

```
docker images -a --no-trunc=false
```

This call lists the entirety of the Docker image library without truncation, which can then be called and utilized using the run variables.

Docker avoids a lot of the dependency loading inherent in the API process, simplifying code and making for a leaner network and system utilization metric. For instance, view a theoretical custom import in Golang:

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    “customlib”
    “main”
    “golang-local/issr”
    “functionreader”
    “payment_processor”
    “maths”
 )
```

In Docker, an equivalent request would simply be:

```
docker run --name tracecrt
```

Caveat Emptor

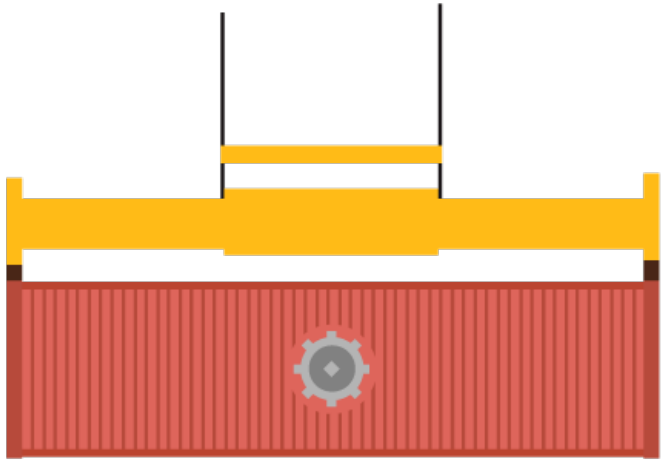
Docker containers are a good solution for a very common problem, but they're not for everybody. While it significantly simplifies the API system at runtime, this comes with the caveat of an increased complexity in setting up the containers.

Additionally, because containers share kernels, there's a good deal of redundancy that is lost by design. While this is good in terms of project scope

management, it also means that when there is an issue with the kernel, with an image, or with the host itself, an entire ecosystem is threatened.

One of the biggest caveats here is actually not one of Docker itself, but of understanding concerning the system. Many developers are keen to treat Docker as a platform for development rather than for what it is — functionally speaking, a great optimization and streamlining tool. These developers would be better off adopting Platform-as-a-Service (PaaS) systems rather than managing the minutia of self-hosted and managed virtual or logical servers.

Analysis: Use Docker in the Right Scenario



Docker containers are incredibly powerful, just like the language that backs them. With this considered, containers are certainly not for everyone — simple APIs, such as strictly structured URI call APIs, will not utilize containers effectively, and the added complexity can make it hard for novice developers to deal with larger APIs and systems.

That being said, containers are a great solution for a huge problem, and if a developer is comfortable with managing images and the physical servers or systems they run on, containers are

a godsend, and can lead to explosive growth and creativity.

Digging into Docker Architecture

The advent of cloud computing has changed the way applications are being built, deployed and hosted. One important development in recent years has been the emergence of DevOps — a discipline at the crossroads between application development and system administration.

Empowered developers have been given a new wide set of tools to enable:

- Application lifecycle management with **continuous integration** software like Jenkins, Travis CI, CircleCI and CodeShip;
- Server provisioning with software and metadata using **configuration management** tools like Chef, Puppet, Salt and Ansible;
- Hosting applications in the cloud, whether they use an **IaaS** provider like Amazon Web Services, Google Compute Engine or Digital Ocean, or a **PaaS** solution like Heroku, Google App Engine or any technology-specific offering.

While these tools are usually wielded day-to-day from the command line, they have all sprouted **APIs**, and developers are increasingly building API clients to manage the DevOps workflows at technology companies just as they do within their own products.

Out of this set of emerging technologies, one of them has taken the world of DevOps by storm in the last three years: Docker.

Virtual Containers

Docker is an open source project that is backed by a company of the same name. It enables one to simplify and accelerate the building, deployment, and running of applications, while reducing tension between developers and traditional Ops departments.

Docker **virtual containers** are packaged with everything the application needs to run on the host server and are isolated from anything else they don't need — containers can be moved from one host to another without any changes. Contrary to hypervisor-managed virtual machines, Docker containers are lightweight and quick to start.

Docker also comes with tools to build and deploy applications into Docker containers. Containers can be hosted on regular Linux servers or in the cloud (or pretty much anywhere using Docker Machine).

Docker Architecture

Each Docker setup includes a **Docker client** (typically a command line interface, but Docker also features a Remote API and a **daemon**, the persistent process that runs on each host and listens to API calls. Both the client and the daemon can share a single host, or the daemon can run in a remote host.

Docker **images** are read-only templates from which containers are generated. An image consists of a snapshot of a Linux distribution like Ubuntu or Fedora — and maybe a set of applications or runtime environments, like Apache, Java, or ElasticSearch. Users can create their own Docker images, or reuse one of the many images created by other users and available on the Docker Hub.

Docker **registries** are repositories from which one can download or upload Docker images. The Docker Hub is a large public registry, and can be used to pull images within a Docker workflow, but more often teams prefer to have their own registry containing the relevant subset of public Docker images that it requires along with its own private images.

Docker **containers** are directories containing everything needed for the application to run, including an operating system and a file system, leveraging the underlying system's kernel but without relying on anything environment-specific. This enables containers to be created once and moved from host to host without risk of configuration errors. In other words, the exact same container will work just as well on a developer's workstation as it will on a remote server.

A Docker **workflow** is a sequence of actions on registries, images and containers. It allows a team of developers to create containers based on a customized image pulled from a registry, and deploy and run them on a host server. Every team has its own workflow — potentially integrating with a continuous integration server like Jenkins, configuration management tools like Chef or Puppet, and maybe deploying to cloud servers like Amazon Web Services. The daemon on each Docker host enables further actions on the containers — they can be stopped, deleted or moved. The result of all of these actions are called lifecycle **events**.

Who uses Docker?

Since it arrived onto the scene in 2013, Docker has seen widespread adoption at technology companies. Interestingly, whereas early adopters for most new technologies are typically limited to small startups, large enterprises were quick to adopt Docker as they benefit more from the gains in efficiency that it enables, and from the microservices architecture that it encourages. Docker's adopters include Oracle, Cisco, Zenefits, Sony, GoPro, Oculus and Harvard University.

Docker's growth has been phenomenal during the last three years, its adoption numbers are impressive, and it has managed to attract investments from top-tier venture capital funds.

Where does Docker fit in the DevOps puzzle?

While DevOps has made it easier to provision servers and to master configuration management for developers and ops teams, it can be a bit overwhelming for beginners. The dizzying number of technologies to choose from can be frustrating, and it can invite a lot of complexity into a company's workflow if the purpose of each component is poorly understood.

Docker doesn't fall neatly into one of the categories we listed in our introduction. Rather, it can be involved in all areas of DevOps, from the build and test stages to deployment and server management.

Its features overlap with those of configuration management software - Docker can be used as a substitute for Chef or Puppet to an extent. These tools allow you to manage all server configuration in one place instead of writing a bunch of bash scripts to provision servers, which becomes unwieldy when the number of servers hits the hundred mark. Complexity invariably starts to creep in when upgrades, installations and changes in configuration take place. The resulting Chef cookbooks and Puppet modules then need to be carefully managed for state changes, which is traditionally a shared task between developers and ops people.

Docker's philosophy around configuration management is radically different. Proponents of **immutable infrastructure** love Docker because it encourages the creation of a single, disposable container with all the components of an application bundled together, and deployed as-is to one or more hosts. Instead of modifying these containers in the future (and therefore managing state like you would

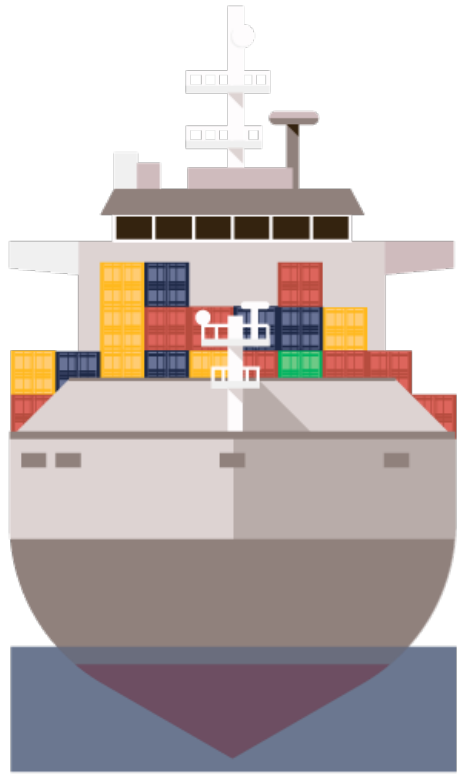
with Chef or Puppet), you can simply regenerate an entirely new container from the base image, and deploy it again as-is. Managing change therefore becomes simplified with Docker, as does repeating the build and deployment process and aligning development, staging and production environments. As James Turnbull writes in The Docker Book, "the recreation of state may often be cheaper than the remediation of state".

Of course, Docker lacks the flexibility afforded by tools like Chef and Puppet, and using it by itself assumes that your team operates only with containers. If this isn't the case and your applications straddle both container-based processes and bare metal or VM-based apps, then configuration management tools retain their usefulness. Furthermore, immutable infrastructure doesn't work when state is essential to the application, like in the case of a database. It can also be frustrating for small changes.

In these cases, or if Chef or Puppet are an important part of a team's architecture prior to introducing Docker, it is quite easy to integrate these tools within a Docker container, or even to orchestrate Docker containers using a Chef cookbook or a Puppet module.

Continuous integration software like Jenkins can work with Docker to build images which can then be published to a Docker Registry. Docker also enables **artifact management** by versioning images. In that way the Docker Hub acts a bit like Maven Central or public GitHub artifact repositories.

Up Next: Tooling Built on Docker Remote API



All of the events listed in the previous section can be triggered via the Docker command line interface, which remains the weapon of choice for many system engineers.

But daemons can also be accessed through a TCP socket using Docker's Remote API, enabling applications to trigger and monitor

all events programmatically. Docker's API also exposes container metadata and key performance metrics.

The Remote API is essentially a well-documented REST API that uses an open schema model and supports basic authentication of API clients. The availability of this API has opened the door for creative developers to build tools on top of the Docker stack. In the next chapter, we'll explore awesome tools that consume the Remote API to see real world examples of API-driven DevOps in action.

Tools Built on Top of the Docker API

As we've seen in the two previous chapters, **Docker** is a popular new technology that lets development teams bundle applications in virtual containers to easily build and deploy them. Docker reduces the complexity of DevOps workflows and encourages the practice of immutable infrastructure, where the entire application, along

with its underlying operating system, is recreated and redeployed as a lightweight container for each change to an application, rather than relying on incremental updates.

Given Docker's remarkable success story over the last three years and the availability of its remote API, it was inevitable that Docker would become a popular platform as well, on which developers have built all kinds of software.

In keeping with Docker's own philosophy, third party developers in this new **ecosystem** have contributed many open source projects. In this chapter we review these projects to see how they are using the Docker API.

Dogfooding

The foremost user of the Docker API is Docker itself — they host a series of tools to combine and orchestrate Docker containers in useful configurations. [Docker Compose] (https://docs.docker.com/compose/) facilitates the deployment of multi-container applications, while Docker Swarm allows the creation of **clusters** of Docker containers.

While Docker itself is active in this area, they welcome the contribution of other actors in **orchestrating** Docker containers. Orchestration is a broad term, but we can break it down into **scheduling**, **clustering**, **service discovery**, and other tasks.

It is usually undesirable to have several processes running inside the same Docker container, for reasons of efficiency, transparency, and to avoid tight coupling of dependencies. It's much more practical for each container to remain limited to a single responsibility and to offer a clearly defined service to the rest of the infrastructure. A complete application therefore usually involves a collection of Docker containers. This introduces complexity for which new solutions abound.

Scheduling

Scheduling of Docker containers is also an important aspect of container orchestration. In this context scheduling refers to the rules by which containers are run on a given host. For example, a scheduling policy might involve two containers having to run on the same host when their functionalities are synergistic. Thanks to this policy, the two containers can be abstracted away as a single application when defining cluster behavior.

In addition to the scheduling features of Docker Swarm, Fleet by CoreOS and Marathon are examples of open source Docker schedulers.

Cluster Management

Clustering involves managing collections of Docker hosts into a cohesive whole so that they might operate together as a single system.

There are open source alternatives for Docker Swarm, like Google's Kubernetes which lets teams define 'pods' of Docker containers. Others include Shipyard, Fleet by CoreOs and Marathon.

Companies such as Spotify have developed and open sourced their own Docker container management system, such is the need for a well-adapted Docker-based system for each use case.

Service Discovery

Service discovery refers to the discovery of the IP addresses related to a service on a network, which can be a complicated process in a multi-host, clustered environment.

Several companies like GliderLabs have leveraged Docker's Remote API to listen to events and create utilities around container-based software. Registrator, an open source project supported by Weave, helps improve **service discovery** by inspecting newly created Docker containers and registering the new services in a directory like Consul.

Networking

To connect the Docker containers that form an application, Docker has some networking features. A default virtual bridge network is enabled by default, but there is a choice of configurations. Vendors have offered alternative network configurations for different use cases.

Weave creates a virtual network of micro-routers to connect containers across multiple hosts. This leads to simpler network configuration, dynamic addition of nodes to the network, and encrypted communications. Weave offers additional services such as the network monitoring product Weave Scope.

Flannel, an open source virtual networking solution by CoreOS, creates an overlay network using an Etcd cluster to store network configuration.

Project Calico is another open source networking solution for Docker containers, based on a pure 'Layer 3' approach, meaning that it is not an overlay network — no packet encapsulation happens above the third layer of the OSI model. This has the benefit of increasing performance when compared to the other solutions.

Storage

One problem that aficionados of **immutable infrastructure** have, which Docker mediates, is databases. Data in databases is mutable by definition, so a container featuring a database cannot simply be recreated from scratch and redeployed without compromising the database's integrity.

There are also native Docker solutions to this problem in the form of data volumes and data volume containers. Data volumes are persistent and survive the deletion of the containers they are connected to, and the data they contain remain inviolate throughout the Docker life cycle. A data volume container can be used when sharing data across multiple containers.

Data volumes can be backed up and restored; products like Flocker by ClusterHQ, an open source data volume manager, manages data volumes and containers, and performs data migration in order to support container-based production databases.

Continuous Integration

Many tools exist in the area of **continuous integration** (CI) to better handle Docker containers in the build, test, and deployment cycle. For example, CodeFresh can be used to trigger the creation and deployment of Docker containers by detecting changes to a Git repository or a new build from the continuous integration server.

CodeShip's Jet is a new CI platform for Docker. It can pull images from any Docker registry and integrate with Docker Compose to easily perform concurrent building and deploying of container-based applications.

Drone is another continuous delivery platform built on top of Docker, that uses an ephemeral container during the build process.

Hosted Docker Registries

In addition to the DockerHub itself, several companies offer [hosted Docker Registries] (http://rancher.com/comparing-four-hosted-docker-registries/), like Quay.io, Artifactory and Google's Container Registry. These services serve as private repositories for containers and offer advanced repository features, third party integrations, and a clean user experience for DevOps engineers.

Log Aggregation

Logspout is another open source project by GliderLabs. When several Docker containers share a single host server, Logspout performs **log routing** and aggregation into a log management system like PaperTrail. Also, Filebeat tallies a container's logs and sends them to Logstash.

Monitoring

A slew of third party **monitoring solutions** for Docker-based apps are available, built by some of the biggest names in the cloud monitoring space. Leveraging Docker's stats API, they display the resulting data in elaborate dashboards.

Examples of these monitoring solutions include:

- A Docker extension by AppDynamics,
- a Docker-enabled DataDog agent,
- first-class citizen treatment of Docker containers in New Relic,
- and Scout's Docker agent, itself distributed as a Docker image.

Configuration Management

Docker lets you add custom metadata to images, containers, and processes (daemons) via **labels**. Labels are key-value pairs that are used to specify custom configuration like versioning and environment particulars.

To avoid naming collisions, Docker encourages the use of namespaces in these labels, but doesn't enforce them. Docker Label Inspector is a utility by Gareth Rushgrove, a senior engineer at Puppet Labs, that checks published Docker images against these guidelines or a given JSON schema.

Security Auditing

As some have raised questions about the inherent **security** of container-based applications, and although Docker itself has plugged many of the holes over the years, vendors have offered solutions to further secure Docker apps. One example is Scalock, a company that has developed software to scan containers for security issues, control access to containers, and monitor containers at runtime to make sure they don't overstep their authorization profile.

PaaS

The multitude of new Docker-based software has arguably engendered new difficulties in composing these tools together, but this is a symptom of a maturing ecosystem. A handful of companies have made the ambitious bet to create an end-to-end PaaS-like solution for building, deploying, orchestrating, and monitoring Docker-based applications, thereby hiding all of this complexity.

Openshift is RedHat's PaaS solution built on top of Docker and Kubernetes. Deis is a Heroku-inspired PaaS solution based on Dockerand CoreOS by Engine Yard. Paz is an open source project based on Docker, CoreOS, Etcd and Fleet that lets teams host their own PaaS-like workflow for container applications.

Lastly, Docker has recently acquired Tutum, a PaaS-like suite for deploying and managing Docker containers, renaming it Docker Cloud.

Full-blown OS

Such is the popularity of Docker that some people have ventured so far as to create entire operating systems made of Docker containers.

Rancher's RancherOS is a lightweight operating system made exclusively out of Docker containers. It is a 20 MB Linux distribution designed to run container-based apps.

HypriotOS is a Docker playground for Raspberry Pi. It allows you to deploy container applications on these tiny computers and consists of a lightweight image that can fit on an SD Card.

Analysis: Remote Docker API Demonstrates Automated DevOps in Action

Docker's emergence has given birth to a new ecosystem of DevOps software. It will likely keep growing in the future as the share of container-based applications (as opposed to VM-based) increases.

Description-Agnostic API Development with API Transformer

Whilst many aspects of the API economy are subject to discussion and conjecture, if there is one truth it’s this: When it comes to successfully describing your API, the jury is out on the best tool for the job. As the API economy has grown so has the number of API description specifications, each with their own specific format and supporting tools. The fact that Swagger has been chosen by the Linux Foundation to be the formative standard for the Open API Initiative might indicate that Swagger will become even more prevalent, but given the investment by organizations in a particular specification choice there is unlikely to be a homogenous approach adopted across the industry any time soon.

Alongside this dialectic, the increasing popularity of the API-first

design approach is making API description specifications more fundamental to the development process. API specifications are becoming the product of design, generated by prototyping tools such as RAML Designer, API Blueprint, Readme.io, or Swagger Editor, with both the documentation and a mock-up available before the developer starts coding. This is a departure from the retrospective method of description that has generally prevailed until recently, where the developer annotates code or manually produces a schematic representation of the API to convey their design. In contrast, API-first design strives to optimize the API expressly for its own goals, purposefully ignoring organizational constraints, architectures and programmatic conventions to create an API definition that exactly meets its design intention and requirements. However, the API-first design principle may also result in technology lock-in for development teams, binding developers to a given set of tools and programming languages that support the chosen specification format.

Where API Transformer Fits

The evolution of the API design and development process is one of the reasons why the API Transformer, created by APIMATIC is an interesting development. API Transformer provides a tool, the "Convertron" that makes it easy for developers to convert from one API specification format to another, supporting the majority of popular API specifications including major Swagger versions, RAML, and WADL. API Transformer provides a web UI that a developer can use to convert an existing specification to an alternative format simply by completing the required fields:

However, the real power of Convertron is that it also provides a

simple API, providing the same functionality as the UI in a manner that can be used programmatically. The example below is a cURL command that converts a Swagger 2.0 specification, the Swagger-provided pet store example to RAML:

```
curl -o petstore.raml -d 'url=https://raw.githubuserconte\
nt.com/swagger-api/swagger-spec/master/examples/v2.0/json\
/petstore.json' https://apitransformer.com/api/transform?\
output=raml
```

This functionality makes several interesting use cases possible:

- It allows developers to choose the API specification that makes most sense to them or can be consumed by their favorite development tools, meaning design and development teams have greater flexibility;
- An API provider can also offer API specifications for download from their developer portal in a variety of formats, which may be helpful to the developer community and increase engagement by communicating to them in a way they understand;
- An API provider can extend the coverage of their testing, by testing against generated formats to ensure there are few semantic differences between one format and another, providing greater resilience in the development process;
- Finally, API providers can also easily migrate away from "legacy" description specifications to ones with better support e.g. WADL.

Example Use Case

In order to test the use cases above we took the concept of a developer portal where API Transformer is used to create translations

of an API description. The API provider that owns the developer portal in this use case specifies their APIs using Swagger, and publishes the specification in different formats as as a convenience to the developer community. In this scenario we envisaged this being a facet of the development process, embedded in continuous integration: When code is published to the git repository for the API, a process is executed that creates translations of the Swagger description in several pre-defined formats. The steps in the process are:

- Developers push code changes to the git repository for the API;
- The CI server detects the commit and spins up an instance of the API, checking to see if the Swagger JSON had changed;
- If a change is detected a Gherkin-style test suite is executed against the API;
- On successful completion of the test suite a version of the specification is generated in the alternative formats the API provider makes available. This is staged ready to be pushed in a CDN to populate the portal.

To demonstrate this we've created a working prototype in Python and Jenkins, the configuration being available on GitHub. Firstly, we created a very simple API using Flask-RESTPlus, describing the API using the Swagger-based syntactic sugar that Flask-RESTPlus offers:

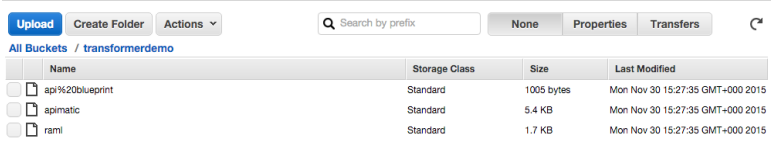
```
from flask import Flask
from flask.ext.restplus import Api, Resource, fields
from uuid import uuid4

app = Flask(__name__)
api = Api(app,
          version="1.0",
          title="API Transformer demonstration",
          description="API created to demonstrate the fun\
ctionality offered by the API Transformer Convertron")
demo_ns = api.namespace('demo', description='Demo operati\
ons')


@demo_ns.route('')
class Demo(Resource):
    @api.doc(description='A demo HTTP GET',
             responses={400: ("Bad request", api.model('E\
rror', {"message": fields.String})),
                        500: "Unhandled exception (captur\
ed in server logs)"})
    def get(self):
        return 'This is a demo!', 200

    @api.expect(api.model('Demo Request', {"data": fields\
.String(required=True)}))
    @api.doc(description='A demo HTTP POST',
             responses={400: ("Bad request", api.model('E\
rror', {"message": fields.String})),
                        500: "Unhandled exception (captur\
ed in server logs)"})
    @api.marshal_with(api.model(
        'Demo Response',
        {"id": fields.String(required=True), "data": fiel\
ds.String(required=True)}), code=201)
```

```
36      def post(self):
37          return {'id': uuid4().hex, 'data': 'Created new d\
38  emo resource'}, 201
39
40  if __name__ == '__main__':
41      app.run(port=8080, debug=True)
```

For the purpose of brevity we then created a simple job that triggered when a commit was made to a local git repository (in reality we would obviously add the test suite and check for content changes for each new version). When triggered, a shell script build step is executed that initializes an instance of our demo API, downloads a copy of the Swagger JSON, and then loops through our target alternative format types:

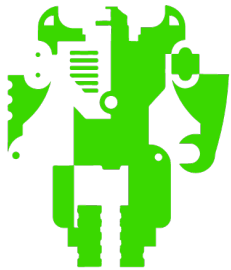
```
1   # Loop through formats and transform
2   for format in raml "api%20blueprint" "apimatic" ; do
3       ext=$(echo $format|tr " " "_")
4
5       # Curl for each target format
6       echo "info: Generating spec for ${format}"
7       curl -X POST "https://apitransformer.com/api/transfor\
8   m?output=$format" \
9       -o $WORKSPACE/transformer_output/app.${ext} -H "Conte\
10  nt-Type: text/plain" -d @$WORKSPACE/swagger.json
11
12      if [[ $? -ne 0 ]] ; then echo "error: Failed to gener\
13  ate spec" && exit -1; fi
14
15  done
```

On successful execution new translations of the Swagger specification are generated in RAML, APIMATIC and API Blueprint formats and saved in the job workspace. The new versions of the specification are pushed to an S3 bucket (using the S3 Plugin, ready to be referenced by a CloudFront distribution.


Upload
Create Folder
Actions
Search by prefix
None
Properties
Transfers
All Buckets / transformerdemo
Name
Storage Class
Size
Last Modified
api%20blueprint
Standard
1005 bytes
Mon Nov 30 15:27:35 GMT+000 2015
apimatic
Standard
5.4 KB
Mon Nov 30 15:27:35 GMT+000 2015
raml
Standard
1.7 KB
Mon Nov 30 15:27:35 GMT+000 2015

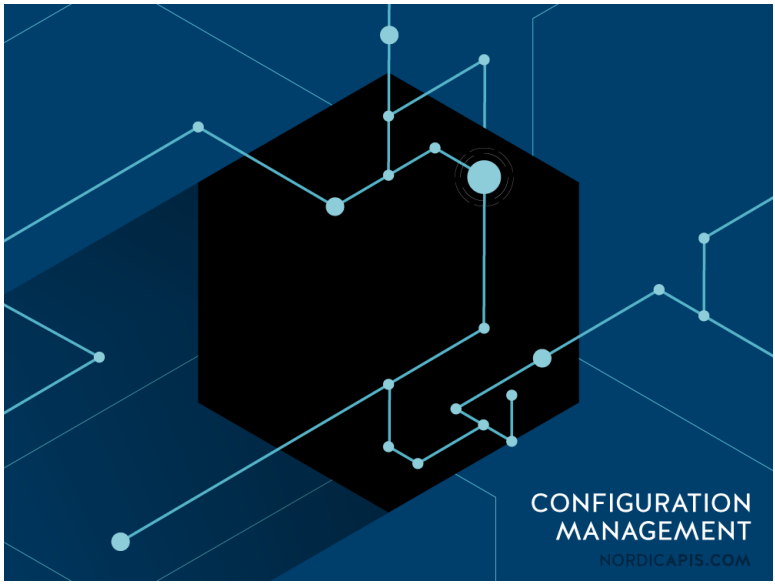
S3 Bucket

Analysis: The Value of API Transformer



There is no doubt that API Transformer offers a useful prospect for API designers and developers to help them quickly convert an API from one description specification to another. Further due diligence and testing on the part of the developer community will ascertain whether any semantic differences exist between source specifications and any of the targets that API Transformer generates, and this will prove its value as a tool for the API community.

The Present and Future of Configuration Management



So far we've described **DevOps** as the set of tools, processes and activities that modern software development teams put in place in order to ensure that they can convert code into live applications in a stable and repeatable way.

It's a bit of a catch-all term, and can be broken down into different areas. One of these is continuous integration, the process of building and testing code at regular intervals to avoid bugs and accelerate integration. Another is **configuration management**

(CM) — the set of practices aiming to manage the runtime state of the applications.

What is Configuration Management?

Every application consists of one or more databases, web servers, application servers, reverse proxies, load balancers, and other moving parts that need to work together at runtime to make a working system.

Software configuration management is the process of describing the ways in which all these inputs — called artifacts or **configuration items** — interact and how they leverage the underlying infrastructure. Specifically, it addresses the installation, configuration and execution of configuration items on servers. Effective configuration management ensures consistency across environments, avoids outages, and eases maintenance.

Traditional configuration management required system administrators to write scripts and hand-maintain files listing technical endpoints, port numbers, and namespaces. When the complexity of a system increases, so does the configuration management process. Artifact versioning, environment changes and tensions between developers and system engineers have led to increased **infrastructure automation**.

CM in the Cloud

The advent of the cloud meant that servers were moved out of on premise data centers and into those of cloud hosting vendors. While the inherent complexities of running an on-premise infrastructure disappeared, new problems arose as well.

Cloud technologies have enabled teams to deploy software to hundreds if not thousands of servers concurrently to satisfy the demands of software usage in the internet age. Managing that many servers requires automation on a different scale, and a more systematic approach. This is where an API-driven approach to CM can help — rather than installing and launching scripts on each node, a centralized CM server could control all of the nodes programmatically and drastically reduce the workload of the team's sysadmins.

New CM software tools have gradually been introduced in the last decade to address these growing needs. In the next two sections we'll look at some of the most popular among them.

The Leaders: Puppet and Chef

Puppet and Chef are the most mature and the most popular CM tools at the moment. The packaging and deploying of applications used to be the sole province of system engineers. By enabling developers to take part in this process, Puppet and Chef have together defined a new category of CM solutions — **infrastructure as code**.

Both are open source projects and based on Ruby (although significant portions of the Chef architecture have been rewritten in Erlang for performance reasons). They both have an ecosystem of plugin developers as well as a supporting company offering enterprise solutions. Each of them features a **client-server architecture**, with a master server pushing configuration items to agents running on each node.

Puppet

Puppet by Puppet Labs was founded by a system administrator named Luke Kanies in 2005. It is built in Ruby but offers its own JSON-like declarative language to create 'manifests' — the modules in which configuration items are described, using high-level concepts like users, services and packages. Puppet is "Model driven", which means that not much programming is usually required. This makes Puppet a hit with system engineers with little experience in programming.

Puppet compiles manifests into a catalog for each target system, and distributes the configuration items via a REST API, so the dev teams don't need to worry about installing and running their stuff — all configuration items will automatically be deployed and run on each node as foreseen in the manifests. Instructions are stored in their own database called PuppetDB, and a key-value store called Hiera. Puppet is used at Google, the Wikimedia Foundation, Reddit, CERN, Zynga, Twitter, PayPal, Spotify, Oracle and Stanford University.

Chef

Chef was created by Adam Jacob in 2008, as an internal tool at his company Opscode. Chef is generally developer-friendly and very popular with teams already using Ruby. It lets developers write 'cookbooks' in Ruby and stores the resulting instructions in PostgreSQL. Chef is used at Airbnb, Mozilla, Expedia, Facebook, Bonobos and Disney.

In both cases, a secure API is available to access any object within Puppet or Chef rather than going through the command line interface. For example, a developer can query the API of each of these to find out how many active nodes are present, or build a plugin for their favorite Continuous Integration system to trigger a deployment every time a new version of the code is built.

A healthy ecosystem of developers have contributed numerous plugins and extensions to both Puppet and Chef. Likewise, Puppet and Chef plugins are also available for popular Continuous Integration products servers like Jenkins, enabling direct integration between the CI and CM processes. That way code builds that pass all the required tests can be automatically delivered to target environments without any manual intervention.

The Contenders: Salt and Ansible

While Puppet and Chef dominate the CM landscape, several contenders have emerged to cater for perceived weaknesses in their architecture.

SaltStack

SaltStack is a relatively new CM tool built in Python and open sourced in 2011. Used by PayPal, Verizon, HP and Rackspace, Salt focuses on low-latency architecture and fault tolerance. It features a decentralized setup, small messaging payloads, no single point of failure, and parallel execution of commands for optimal performance.

Ansible

Ansible, also open source and built in Python, was created by Michael DeHaan in 2012. It was created in reaction to the relative complexity of Puppet and Chef, and attempts to offer a simpler, more elegant alternative, with a shorter learning curve.

Contrary to Puppet, Chef and Salt, Ansible is based on an agentless architecture — meaning that no agent is required to run on each infrastructure node, which leads to less complexity and less load

on the network. Ansible modules, referred to as units of work, can be written with a variety of scripting languages like Python, Perl or Ruby. Ansible lets users define playbooks in YAML for often used system descriptions.

Ansible users include Apple., Atlassian, EA, Evernote, Twitter, Verizon, NASA, Cisco and Juniper Networks.

Aside from Puppet, Chef, Salt and Ansible, there are many other CM options, such as Capistrano and SmartFrog. Each one of them differentiates in a certain way. For example, Otter has a web based user interface which lets you switch between a drag-and-drop editor and text mode, along with first class support for Windows.

Cloud Vendor Solutions

Infrastructure-as-a-Service vendors like Amazon Web Services come with their own, highly specific concepts and terminology, and CM tools need to speak that language to let DevOps people navigate their product stack.

All of the above mentioned CM products have extensions for Amazon EC2, and some of them have native support for Google Compute Engine. Amazon has its own native configuration management product called OpsWorks which competes with Puppet and Chef for applications entirely hosted on Amazon Web Services (although OpsWorks itself is based on Chef internally).

Vagrant by HashiCorp is an open source tool to manage virtual development environments (such as VMs and Docker containers) and wraps around CM tools. With Vagrant teams can create portable development environments that can be moved between hosts without any changes to the configuration.

In-House CM tools

Some companies have very specific requirements around CM that are hard to meet by using one of the available products on the market. Building a custom, fully-fledged configuration management solution represents a great deal of work, but it can be worth it for companies that have the means to see it through.

Netflix created their own configuration management API called Archaius. Named after a species of chameleon, this Java and Scala-based in-house tool lets Netflix perform dynamic changes to the configuration of their Amazon EC2-based applications at run time. It was open sourced in 2012.

Netflix had a variety of reasons for building an alternative to Puppet or Chef. High availability is paramount to their business model, so they can't avoid any downtime related to server deployments. All of their applications are designed in such a way that configuration can be reloaded at run time.

In addition, Netflix servers span multiple environments, AWS regions and technology stacks, which are collectively called 'context'. Thanks to Archaius, Netflix are able to enable/disable features dynamically depending on their context.

Analysis: The Road Ahead

While the configuration management technological landscape has greatly matured over the last few years, there is general consensus that things will keep evolving in the future. The current mainstream solutions are often viewed as too complicated, unforgiving and a hassle to maintain. One alternative to the classic CM approach is the emerging **immutable infrastructure** championed by (Docker)[http://nordicapis.com/api-driven-devops-spotlight-on-docker].

Whereas regular configuration management aims to define and manage state at run time, containerized applications require that all the configuration be defined at build time. The resulting portable containers can then be moved from one host to the next without any changes in state.

These states can then be saved in Docker images and used later to re-spawn a new instance of the complete environment. Images therefore offer an alternative means of configuration management, arguably superior to runtime configuration management.

Another alternative is to hide the complexities of configuration management with PaaS solutions like Heroku that deal with CM under the hood and packages everything needed to run an application in buildpacks. While less flexible than IaaS, it offers the luxury of ignoring the CM process entirely.

It's unclear where configuration management is headed, but one thing is certain — it will remain one of the chief concerns of all DevOps teams for the foreseeable future.

Security for Continuous Delivery Environments

Continuous delivery is a hallmark of the modern development world. As tools have matured and the needs of the consumer have evolved, constant development and deployment have become the norm rather than the exception.

With this increase in deployment, security has increased part and parcel. In this chapter, we're going to discuss how to maintain security in such a unique deployment environment, and the challenges inherent therein.

What Is Continuous Delivery?

Continuous delivery is the process by which developers push consistent and timely updates via a deployment system. This is typically an automated system, wherein DevOps teams join ideation, initial development, and deployment into a single, agile development track.

There's a lot to be said for this kind of delivery interaction. For one, the system is far more agile to the needs of the market — because delivery isn't tied to a long-term cycle, features can be rapidly developed and pushed through quality assurance as consumers notify the developer of their needs.

This cycle change also means that when errors and bugs arrive,

they're typically short-lived. Developers can rapidly address security concerns, bugs, and errors through additional patching and deployment, reducing the effective life of issues in an API.

As part of this change to an automated and continuous development cycle, there comes some caveats that prohibit more traditional development. Most importantly, the common practice of manual code auditing becomes unrealistic due to the sheer rapid agility of development.

Not everything is "sunshine and rainbows", though. Rapid and continuous delivery has some caveats that developers need to manage.

Chief of these is the fact that rapid and continuous development can make feature creep easier to engage in. With ongoing incremental releases, the "greater picture" is often lost, and feature creep becomes a legitimate issue. Likewise, constant continuous deployment can also proliferate bugs that would otherwise be eliminated over long-term testing and implementation.

These caveats are nothing compared to the benefits granted by increasing agility and consumer interaction, but they allow a unique perspective on development — continuous deployments inherently require more consistent integrations — all of which need to be secured properly.

Auditing Security

Thankfully, there are a number of ways an API provider can audit and secure their APIs in a continuous delivery environment. While each of these solutions are incredibly powerful, they are generally best used for specific use cases — there is no "perfect" implementation.

Code Scanning and Review

Code scanning — the automated process by which code is scanned and checked for vulnerabilities — is an incredibly powerful tool for auditing security. One of the most powerful features of code scanning is the fact that, in most solutions, the code is checked against common and known vulnerabilities, removing a lot of the dependency based issues that plague rapid codebases.

Implementing this as a development procedure makes sense, but even so, it's often overlooked. When you submitted a final paper in school, was it a first draft? Of course not, most students passed the work through spell-check a hundred times, checked the grammar, checked every single fact, checked their periods and commas, and made sure everything flowed.

Accordingly, knowing how many people depend on the functionality within, why would an API developer release a product without first doing their own "spell-check"?

A lot of these solutions are additionally open source. While there's been a lot of discourse about open source security, and whether or not it's actually as powerful and useful as has been stated, the power of collaboration makes having a crowd sourced, open database of faults more powerful than having a closed, limited list of possibly outdated and irrelevant references.

Creating a Clean Ecosystem

Code scanning can only go so far, however — for many development teams, the devil's in the details. Establishing a secure and stable development and operations platform is just as important as scanning code for common issues.

There seems to be a disconnect in most DevOps systems where the development and operations clusters are far removed from one another. What this ultimately results in is a system where hotfixes

are applied to properly functioning code on one cluster to get it to work on another cluster.

While this is fine for the basal requirement, it's terrible for security, as it often introduces new, unique errors and faults that would otherwise not exist without this cluster discrepancy.

As crowdsourcing has become more accepted by the mainstream, there have been more and more tools introduced to the market that harness the power of the group to produce some amazing results.

One such tool in the security space, Evident.io, utilizes crowd-sourced environment and protocol registers to intelligently analyze code, reducing complexity to understand analytics. These analytics are then used to pinpoint attack vectors, expose common issues, and clarify security issues that can be hard to see.

Adopting More Effective Development Strategies

The adoption of two-speed IT as a production principle is also incredibly powerful for both production and security. In this approach, two "lanes" are formed — rapid beta development and static release development.

In this approach, the rapid beta development is where new features are crafted and implemented, whereas the static release development track focuses on releasing products that meet need requirements and are stable.

Positioning separate tracks helps ensure security in a continuous environment as it allows for an opt-in channel for experimental and beta features without impacting the stable track. The security for the opt-in track does not necessarily need to be as intense as the stable track, as the de jure principle is certainly "caveat emptor".

That being said, implementing future features in a low security environment can help pinpoint the holes in the armor that might

otherwise be obscured when implemented in a high security environment.

Segmentation of Services

While creating a "unified" experience for developers has long been the rallying cry of most API proponents, in some cases, it is actually better to segment services, especially in the case of security and auditing.

Consider the following example. An API provider has created a "unified" API that combines data processing, media conversion, and large file transfer between servers, clients, and users. Each update to code requires a long-term audit, with multiple teams using the same code base.

What are the problems with this application? Well, first of all, we have multiple teams utilizing the same general codebase and applying specific solutions therein. The best operations schematic for the Media Conversion Team may not necessarily be best for the Data Processing Team, and certainly not for the Large File Transfer Team. With each new code fix, the code bloats, and different teams implement solutions that are contradictory in nature. Even with the teams conversing directly, this is inevitable.

What's the solution? **Segmentation**. With segmentation, developers take functionality and divide the API along those lines. Essentially, a "main" API is developed to unify the functions in these other, disparate APIs, allowing individual APIs to be formed for specific use cases and functionalities.

In such a development process, the API, which formerly looked like this:

- **Function API** - Media Conversion, Data Processing, Large File Transfer

Turns into this:

- **Function API** - API with general purpose calls, tying into:
- **Media Conversion API** - API specifically designed to convert media for use in either Data Processing or Large File Transfer;
- **Data Processing API** - API specifically designed for large data processing for use in either Large File Transfer or Media Conversion;
- **Large File Transfer** - API specifically designed to handle the transfer of large files, including those generated from the Media Conversion and Data Processing APIs;

By segmenting the API into various secondary APIs, each essentially becomes its own development segment. By doing this, security can be audited for each function, as the security needs of each is drastically different.

Most importantly, segmentation results in secondary layers of security. This creates a situation where, even if a hacker can break through the "Function API", additional gateways for each new segment makes it almost impossible to actually get through the security ecosystem.

Analysis: Continuous Delivery Requires Continuous Security

Continuous Delivery is an incredibly powerful implementation, but it comes with its own issues and security concerns. While ensuring users have the most up-to-date revisions of a code base can make for more powerful interactions with that code base, it can also necessarily increase the chance of code failure or lax security. The solutions offered here are but few of the many solutions which can be implemented to negate the concerns offered by the development strategy.

While adoption of some or even all of these security solutions might seem a daunting prospect, the fact is that most API developers should implement them regardless — nothing but good can come from proper design approaches, segmentation, and code scanning.

Implementing Continuous Delivery is not only one of the best solutions for API developers facing large development lifecycles — it's possibly the most powerful method for forming a strong userbase and ecosystem.

API Testing: Using Virtualization for Advanced Mockups

Simulated environments are not a foreign concept in web development. Especially for application programming interfaces — APIs — that may need to create a simulation of their service for **testing** purposes, virtualization is an option that can go beyond your average API explorer or GUI.

Following the Dropbox API explorer enhcnacement release, now Amazon has recently announced a mock integration feature to their API gateway. Virtualization takes mock testing a step further, allowing API calls and simulated responses to be coded into early stage app development, enabling both API providers and API developers to gauge performance in quite fine-grained ways prior to an official API launch.

And as web API development becomes more iterative and constantly in flux (According to Michael Wawra of Twilio, your API is never really finished), some providers are pushing **simulation** as the solution to help keep an entire API platform agile.

What is Service Virtualization?

In the programming world, Service Virtualization is a method of abstracting the behavior of cloud driven applications. "Mocking on steroids," virtualization doesn't just mock a specific function, it emulates the same performance as an end product would. Developer operations can use virtual services to begin functional, integration, and performance testing early on, rather than after an official product launch.

Why virtualize a product? For physical products it may mean avoiding a major recall. In his talk with Nordic APIs, Matti Hjelm references LEGO's failed Fun Snacks. Made in the shape of lego bricks, parents became furious with LEGO for teaching kids to eat things that look like identical to the normal LEGO bricks.

In hindsight, LEGO's Fun Snacks lacked the proper **user testing**. For web services, virtualization can similarly help with that quality assurance aspect and simulate actual behaviour, capture information, and use feedback to replace or change components — hopefully to avoid the choking hazard for developers and end users.

What is API Virtualization?

API virtualization allows you to isolate components and simulate external, non-network based APIs. This runtime behaviour simulation uses a tool in place of the real mcCoy — a virtual copy of your API that mirrors behaviour of the final production version.

In web software, an error-free finished product is a daydream. Instead, rapid (and continuous) development cycles occur that need immediate testing. Next we'll outline five benefits that API virtualization can bring to the table.

1: Test Failure in a Safe Environment

Want to see how error messages function? How rate limits work? From a user's perspective, a virtual API looks and behaves like a real service. However, distanced from live runtime, virtualization can be used for **simulating** drastic scenarios. Use an emulator to simulate real world behavior like **downtime**, slow or erratic API responses to see how an app behaves when confronted with these dilemmas.

A great test can provide the information on what happens when a client calls an API that suddenly responds strangely, and do so in a neutral, risk-free setting.

2: Increase Development Productivity

Dependency injection is a messy situation, and writing mocks are useless for developers in the long run. API virtualization allows the ability to reduce redundancies in the development cycle and emphasize a more seamless continuous integration process.

Desktop Software Maturity Stages

According to Matti Hjeml of SmartBear, typical desktop software can be said to have a development lifecycle made up of these stages:

- 1: The technology stage. You develop your core functionality with a simple, non-cluttered User Interface (UI).
- 2: To stay ahead of competitors or in response to customer inquiries, features are added. Design may become 2nd priority in this stage, leading to UI clutter.

- 3: Customers ask for simplicity in response to a cluttered UI.
- 4: Software becomes a commodity, and integrates with other software, plugins, or other tools through the use of embeddables, plugins, SDKs, etc.

API Market Stages

Hjmel sees a very similar development cycle within the API market:

- 1: Unique feature set, hard to use?
- 2: Add features to keep competitors behind
- 3: **Improve DX to keep 3rd party devs coming back**

A significant aspect of stage three, improving developer experience, is serving the API with the proper **testing environments**. Whether with interactive documentation, sandbox, or virtual API, having a way to test an API is an important cog to complement your developer ecosystem.

3. Isolated Performance Testing Saves Money

In normal app development practice, you may simulate other APIs, but in the process become too independent, using mock code that mimics the API that you don't have control of yourself. By virtualizing your *own* API, API platform providers can erase these type of headaches, and allow more sharing with the API developer users. However, this must be done with tact.

Share Testing Environment

Why don't you simply share a testing environment running on internal servers with the user? According to Hjelm,

> "The problem is that in this new economy user needs are unpredictable. It's not like the SOA world where you had control of which were the users and when they were allowed to use your system… There is no predictability when and where the API calls are coming from and how much."

If general usage is unpredictable, most likely the testing is unpredictable as well. The bandwidth cost for large random testing can be a strain on an API provider's server. So…

Encourage Local Testing

To combat this problem, Hjelm encourages providers to "build shareable and distributable virtualized versions of your API." This can simplify testing locally, and empower developers with autonomy and control. It still shows an active engagement from the platform owner in third party testing, integration, and development.

Running test servers can be costly, and demands for performance testing are largely unknown in early stage development. If the micro costs incurred for API calls are adding up, an API developer could **save money** using a virtualized API, rather than performance testing on the production version.

4. Reduce Time to Market

APIs are distinct from normal products, as potential revenue relies on a long, two-product lifecycle. First is the API product lifecycle itself, in which one must plan, develop, and form an ecosystem of developer consumers. After this tremendous feat, APIs rely on developers to create successful apps before a significant quantity of calls are made (and income generated for the API provider).

By using virtualization techniques, however, this cycle can be significantly cut down. By offering a virtual API in a continuous development mode, third party developers can start building applications **before the actual endpoint is live**. This could significantly reduce the time from when the API is developed to the first call being made by an end user within a third party application.

5. Virtualization Can be Usable

A virtual API must emulate the core functionality of your normal API, and more. It should be able to simulate awkward behaviour, or slow response time — natural occurrences, and respond with typical error messages. It should be easily configurable, supporting security with OAuth 2. As Hjelm says, design your virtual API as data driven, distributable, and deployable locally.

> "By recording and storing known answers to predictable requests, then simulating the service and playing back known ("canned") data, API Virtualization allows build systems to do more, with faster, more predictable results. This does not remove the need for end-to-end testing, but it does allow the team to have more confidence with each build."
— Matthew Heuser

One may think that this is a difficult process, but it can be implemented in a few steps. In a previous post, we used SmarBear's ReadyAPI service to mock the endpoint of a REST API and create a virtual service in about 20 steps.

Safe Harbor: Drawbacks, Risks, and Mitigations

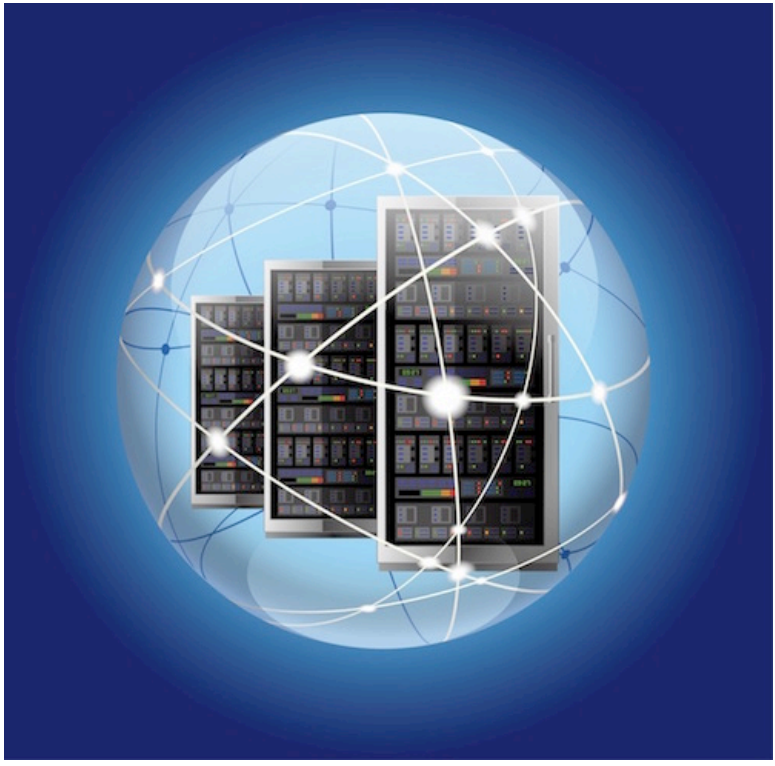
- **Virtual Insanity**: There is the risk that the production API won't match the virtual API that has been used for testing. Teams must have automated checks in place for end-to-end testing when using the production API.
- **Analytics**: How do I analyze usage patterns? You can use in-app Google Analytics.
- **Complex Maintenance**: According to Hjelm, it shouldn't be hard to maintain a virtual API, as APIs deliver simplistic views of complexity.

"Virtualize Me" -Your API

Virtualizing your test environment, and doing so often, and early on, can accelerate rapid development cycles, and increase internal productivity. Some argue that the future will see the emergence of more Sandbox-as-a-services and complex simulation environments. As Kin Lane says:

> "I'm seeing services emerge to cater to this need, as with Sandbox, which is a service layer of the API lifecycle I think will only grow in coming months. I can see the need to create simple sandbox for any APIs as you are kicking tires, but I can also see the need for sophisticated simulation environments built on top of common APIs, allowing some apps to test out more advanced, and specialized scenarios."

Analysis: Benefits in API Virtualization



API virtualization allows quickly curated versions of the API for your developer ecosystem to emulate immediately. To review, the benefits of API virtualization can include:

- Increased overall productivity
- Shorter time to market
- A true environment for third-party sandboxing
- Test real world scenarios without the risk

- Eliminate the need for individual developers to write and rewrite their own mocks

Other Resources on API Virtualization, Sandboxes:

- Walkthrough Creating a Virtual Service Ready API, Nordic APIs
- Patterns of API Virtualization
- What is API Virtualization?
- What is an API Sandbox?
- API Virtualization, Sam Ramji
- I Wish all APIs had Sandbox Environment by Default, API Evangelist
- Virtualized API Stacks, API Evangelist
- Sandbox
- RESTful API Design: API Virtualization, Brian Mulloy | Apigee
- Service Virtualization as an Alternative to Mocking

Automated Testing for the Internet of Things

The Internet of Things (IoT) is upon us. Every object you see around you — whether it's your fridge, your electric toothbrush, your car, even your clothes, are about to acquire a form of sentience.

Some of them already have. Fitbit watches, Nest thermostats and Apple TVs are just the tip of the iceberg when it comes to the Internet of Things. **Sensors, embedded systems and cloud backends** are coming together to bestow smartness upon a bewildering array of previously inanimate, dumb objects. IoT will be a trillion-dollar market by 2020 and every big player in the hardware space is jockeying for position, along with a flood of new IoT startups.

While embedded systems have been around for a very long time in the form of consumer electronics, IoT has given them a new dimension. Previously these systems were essentially self-contained and could work in isolation. But **connected objects** now need to

converse with each other and rely on each other. Developers have had to start thinking about device-to-device (D2D) and device-to-server (D2S) communication and of course human interaction that comes into play as home appliances and a host of other everyday objects essentially become an extension of the Internet.

IoT and Testing

In traditional software development, code can be built and tests can be automated in production-like environments. The modern approach that makes this process repeatable and predictable is called Continuous Integration. Its purpose is to promote code quality, catch bugs early, reduce the risk of regressions and accelerate development iterations. It is very mature in the web development space, and increasingly so in mobile app development as well. In fact, test automation is so ingrained in the mind of developers that many have changed their entire approach to programming, favoring **test-driven development** — a paradigm where testing drives code design and development, rather than the other way around.

Though things are increasingly complicated in the embedded world, the end user's expectations are the same — modern systems should just work. As the recent Nest thermostat dysfunctions have taught us, IoT products are not yet as robust as their more traditional counterparts. To fix this IoT vendors are moving to integrate better continuous integration into their daily practices.

Until recently continuous integration had never been a fixture of embedded software development, mostly because the interplay between hardware and software made things more difficult.

What Makes IoT Applications Harder to Test

Agile methodologies require a different approach when hardware is involved. More up front design is required, and discarding previous iterations of the product comes at a greater cost than in purely software projects. At the very least, iteration times are longer.

One assumption that needs to be verified in order to achieve continuous integration in the IoT is the possibility of **test automation**. While it can be achieved in the embedded space, a number of hurdles need to be overcome. It isn't as easy to isolate code because of the dependencies to the underlying hardware that can hardly be overlooked.

IoT systems are composite applications that need:

- The ability to gather sensor data by reacting to a variety of inputs like touch, voice and motion tracking
- Different types of interoperating hardware, some of them well-known like Arduino boards or Raspberry Pi, others more unusual or context-specific, like a smart coffee machine, video camera or oven
- Cloud-based servers, mobile and web applications from which the devices can be monitored and controlled
- A Device API to enable routine data and device diagnostics pulls to cloud servers, as well as functionality manipulation.

To complicate things even further, IoT comes with its own protocols like MQTT, CoAP and ZigBee in addition to Wi-Fi and Bluetooth. Furthermore, embedded systems are subjected to **regulatory requirements** such as IEC 61508 and MISRA to ensure the safety and reliability of programmable electronic devices.

Programming languages used in embedded systems tend to be either C or C++. These languages, more low-level than those used in

web development, typically imply better runtime performance but longer programming lead times and more hassle due to the need for explicit memory management, not to mention less available talent. Lack of code portability means that cross-compilation is required between development and target environments.

Independent **Greenfield projects** are relatively rare in embedded software — projects often have dependencies on monolithic legacy code into which CI principles are difficult to retrofit. Similarly, subsystems of IoT applications are often owned by different vendors. What happens if a bug is found in a vendor-supplied device that the application setup depends on?

Exhaustive test data, including telemetry data, can be difficult to obtain for IoT projects since it depends on real world situations where things like weather conditions and atmospheric pressure can influence outcomes.

Finally, non-functional requirements tend to be difficult to test against in systematic ways. These requirements can revolve around bandwidth limitations, battery failures, and interferences.

Simulations, Quality Assurance and other Approaches

Despite the problems listed in the previous section, teams building the future of IoT are trying to achieve systematic, repeatable, and regular release processes in the embedded world.

Part of the guesswork around test planning can be limited by making familiar choices where possible in the architecture. One example is to use Linux for all embedded software projects, so that at least the **operating system** layer remains predictable. In general, the ability to decouple programming logic from hardware makes for easier subsystem testing. Also, focusing early efforts on prototypes

or low-volume first iterations can help chip away at the guesswork and ease further planning.

To mitigate problems around test data quality, IoT professionals record input data from real world users and environmental data, to be replayed in test environments. This helps to keep test environments as production-like as possible. Compliance testing against regulatory requirements can be partially automated using static analysis tools. DevOps automation solutions like Electric Accelerator include these types of checks out of the box.

But mostly, modern approaches to testing composite applications featuring embedded software involve some form of simulation. Much like the way mobile app developers use emulators like Perfecto and Sauce Labs to recreate test conditions across a variety of smartphones, embedded software developers resort to simulation software to abstract away parts of their continuous testing environments. In particular, simulations resolve the problems of hardware availability and accelerate tests against various versions, screen sizes, and other hardware properties.

These virtualized environments are specific to each application and tend to be expensive to set up, but as is the case in traditional continuous integration, the initial effort pays back for itself many times over as the project's life extends. Not only do they afford much more flexibility in test setups and scale, but they help put testing at the center of the team's preoccupations.

A hardware lab will usually follow the **Model-Conductor-Hardware** (MCH) design pattern. The **Model** defines the abstract logic underpinning the system, holds the system's state and exposes an API to the outside world. The **Conductor** is a stateless component that orchestrates the bidirectional interplay between the Model and the Hardware. The **Hardware** component serves as a wrapper around the physical hardware. It triggers or reacts to events by communicating with the Conductor.

In a virtual hardware lab, the Hardware and the outside world (and

its relationship to the Model) are replaced by a simulation so that the entire setup becomes purely software-based. A small number of vendors have built simulation offerings like Simics by WindRiver, which offers virtual target hardware that can be modified or scaled at will.

These simulators are becoming increasingly sophisticated, supporting performance and memory testing, security testing, and they even allow for edge case tests and fault injection, like dropped connections or electromagnetic interference creating noise in sensor data.

Nevertheless, a decent **quality assurance** process using real hardware is still necessary at the tail end of each major testing cycle. There are several reasons why teams may never rely entirely on simulations. For one thing, errors and approximations in the simulation can cause imperfections that must be caught before shipping. In addition, human interaction testing and emergent behaviors can't be simulated, and a person's emotional response to even a perfectly functional hardware-enabled system can be difficult to predict. A combination of white-box testing and black-box testing is usually employed during the QA phase to detect problems that might have fallen through the simulation's cracks.

New Frontiers

In the age of the Internet of Things, slow testing cycles and poorly tested products are no longer sufficient. Companies have adapted their internal processes to reflect these new expectations to thrive with products blending state-of-the-art software and hardware.

Smart objects like unmanned aerial drones will be subjected to deep scrutiny and regulations, and users will become less accepting of glitches in their smart home appliances. More companies will offer IoT-specific testing software, like SmartBear whose Ready! API

solution enables API testing with support for MQTT and CoAP. As a side effect, test automation job opportunities will likely increase in the embedded world, creating new career prospects for engineers who straddle the line between software and hardware.

Expectations around new software deployment and availability of new capabilities on existing hardware have been greatly increased by recent advances in mobile and automotive firmware delivery processes. Until recently, buying any electronic device constituted a decreasing value proposition — the device would gradually lose value over time and consumers would be pressured to buy new versions of the hardware to benefit from new features.

But **Over The Air (OTA) updates** of firmware has changed that. OTA is a deployment method where software updates are pushed from a central cloud service to a range of devices anywhere in the world typically via wi-fi but also via mobile broadband, or even IoT-specific protocols like ZigBee.

Smartphones were the first connected devices to feature OTA updates, leading to longer-lived devices and a diminished feeling of **planned obsolescence**. Cars came next — Tesla famously designs their cars so that software upgrades can fix problems and enable new capabilities via OTA updates. This requires careful planning and a systematic approach to software delivery. One recent is example is the auto-pilot feature on the Tesla Model S that was made available to existing cars after an OTA update. The cars had already been equipped with all the hardware necessary for the autopilot to function (cameras, sensors, radar), and a pure software update was then enough to enable the new feature.

The fact that they are able to confidently ship these changes to a product like a car, for which safety and usability are paramount, speaks volumes about the level of planning and test automation that they've put in place. The sensitiveness of these updates will only increase in the era of self-driving vehicles, when artificial intelligence will replace humans at the controls.

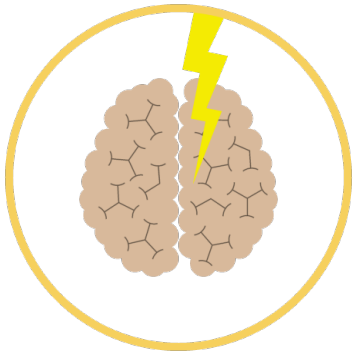
Tesla can't fix everything with OTA updates — it has had to physically recall large numbers of cars for problems in the past, but this powerful new delivery method has given customers the expectation that the product they buy will improve over time.

Analysis

OTA support must be built into all the supported devices, and IoT vendors need to do careful version management and testing in simulated environments in order to achieve seamless delivery in this way. To avoid long roll out times only the smallest possible delta files should be delivered via OTA updates, and security can be an issue as well. Companies like Redbend and Resin offer device management and OTA update workflow solutions to IoT customers.

The Internet of Things promises to change the way we relate to objects all around us, but the high expectations set by companies like Apple and Tesla will require IoT companies to evolve and elevate the culture of testing in consumer electronics to new standards.

Final Thoughts



Continuous integration is part of the agile development methodology that is making the web run more efficiently. Throughout this volume, we've covered how the union of development and operations can help produce more iterative life cycles, allowing both beta tests and launched products to realize their potential. With DevOps tooling for functional testing, code generation, configuration management solutions, and more being programmable via APIs, the entire DevOps process is becoming an *APIOps* process.

We hope you enjoyed this volume. For more, subscribe to our newsletter for a refined digest dedicated to helping you make smarter tech descisions with APIs. We publish new articles on our blog every week, and also hold events that bring together API practitioners to talk strategy.

Hope to connect with you in the future,

- Nordic APIs team

More eBooks by Nordic APIs:

The API Economy: Tune into case studies as we explore how agile businesses are using APIs to disrupt industries and outperform competitors.

The API Lifecycle: An agile process for managing the life of an API - the secret sauce to help establish quality standards for all API and microservice providers.

Programming APIs with the Spark Web Framework: Learn how to master Spark Java, a free open source micro framework that can be used to develop powerful APIs alongside JVM-based programming languages.

Securing the API Stronghold: The most comprehensive freely available deep dive into the core tenants of modern web API security, identity control, and access management.

Developing The API Mindset: Distinguishes Public, Private, and Partner API business strategies with use cases from Nordic APIs events.

Nordic APIs Conference Talks

We travel throughout Scandinavia and beyond to host talks to help businessess become more programmable. Be sure to track our upcoming events if you are ever interested in attending or speaking. Here are some examples of sessions from previous events:

- Introducing The API Lifecycle, Andreas Krohn
- You Need An API For That Gadget, Brian Mulloy

- Pass On Access: User to User Data Sharing With OAuth, Jacob Ideskog
- APIfying an ERP, Marjukka Niinioja
- Integrating API Security Into A Comprehensive Identity Platform
- A Simpler Time: Balancing Simplicity and Complexity, Ronnie Mitra
- The Nuts and Bolts of API Security: Protecting Your Data at All Times

Endnotes

Nordic APIs is an independent blog and this publication has not been authorized, sponsored, or otherwise approved by any company mentioned in it. All trademarks, servicemarks, registered trademarks, and registered servicemarks are the property of their respective owners.

Nordic APIs AB Box 133 447 24 Vargarda, Sweden

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact