WHITE PAPER–APRIL 2020

A Guide to Spring Development on VMware Tanzu Application Service

vmware®

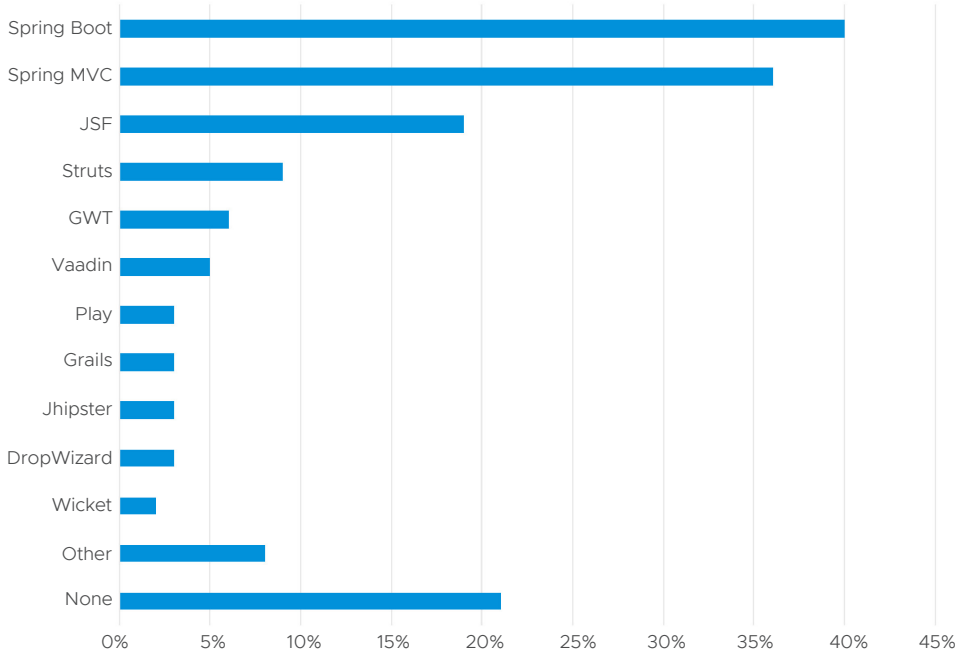
Table of contents

Pivotal is now part of VMware, so some of these products and services are now part of VMware Tanzu™. *Learn more*.

Introduction

"Few words can better express the Spring domination in the Java ecosystem than this graph. With 4 in 10 developers using Spring Boot in their applications, it's interesting to see it has overtaken the Spring MVC framework for the first time."

SNYK AND JAVA MAGAZINE [1]



If you're reading this paper, you're probably using *Spring Boot*. It's a terrific framework for building apps in *cloud-native Java*. By now, you probably have lots of Boot apps. And you might be thinking "where's the best place to run them?"

Don't take this decision lightly. With the right choice, you can achieve supersonic velocity on the Java Virtual Machine (JVM)—the rapid delivery of high-quality software over and over again. In fact, a lot hinges on your ability to move to continuous delivery with Spring Boot. Consider the stakes for your enterprise.

New start-ups are coming after your customers with shiny new tech. The hyperscale web companies are entering new markets all the time. Meanwhile, you're trying to figure out how you can become the disruptor in your industry, and use software to differentiate and gain market share.

The good news is that hundreds of organizations like yours have met this challenge head-on with Spring Boot and *VMware Tanzu Application Service* (TAS). You already know what *makes Boot wonderful*. What's so special about Tanzu Application Service? And why does it run Boot apps so well?

1. Snyk and Java Magazine. "JVM Ecosystem Report 2018."

Objectives, definitions and scope

Think of this paper as a reference guide. You should find value in individual sections, as well as the whole document.

Our goal is to explore the powerful one-two punch of Spring Boot microservices running atop Tanzu Application Service (TAS). Rarely, companies seek to go slower. Rather, that's a side effect of having too much risk and not enough options. The combination of Tanzu Application Service (TAS) and Spring Boot provides you with the least amount of risk with the greatest amount of productivity.

In this paper, we're going to put these claims to the test. How? First, we'll build out a small collection of apps. Then, we'll push every bit of it to TAS, our chosen cloud platform.

So let's learn a bit and have a dose of fun.

What's an application platform anyway?

An application platform is a popular way to run your custom code at scale. TAS is one example of this type of solution. Enterprise development teams use TAS and other application platforms to handle the entire software development lifecycle.

Why the term "application"? Well, with an application platform, the deployable artifact is often a compiled, deployable application library. For Java, you would push a *.jar file; for .NET, you'd push a *.dll or *.exe. So instead of using a container image, you would just push your application to the platform, and the tool would do the rest of the work for you.

What do we mean by the rest of the work? Well, a modern app platform will do these things on your behalf:

- Network and routing – Load balancing, DNS and other networking functions should be done for you
- Logs, metrics and monitoring – Application telemetry to help you observe and troubleshoot your apps
- Services marketplace – A structured way to add backing services to your apps
- Team, quotas and usage – Useful governance features that help you adhere to common enterprise tenancy requirements

All of these elements round out a complete experience for development and operations teams—hence the platform moniker.

Does a good application platform use containers? You bet. In fact, app platforms will build the container for you. In the case of TAS, it uses the popular buildpack model to containerize your app. This approach helps the developer because you don't have to fiddle with the container lifecycle. (As it turns out, you can *run your own Docker or OCI-compliant container images on TAS if you want to*. But that's a *story* for another time.)

At the end of the day, TAS lets you focus on writing code that runs well at scale.

Now that you know a bit more about application platforms, let's shift gears to talk about microservices.

Microservices under the microscope

Before we dig into the code, let's unpack microservices a tad. This pattern is essential for your success developing cloud native software.

Microservices don't imply a framework, language or runtime per se. You have a lot of freedom. Given that, it's easy to see how a small development team—a start-up—can manage microservices. But what about an enterprise? Can microservices really scale with thousands of developers? It's not hard to imagine microservices resulting in a macro mess.

Microservices at enterprise scale

Microservices at scale are worth considering if you need an architecture that's stable, flexible, scalable and secure. Consider asking *these questions* before you opt for microservices:

- Does your system need to evolve at different speeds or directions?
- Does a module have a completely independent life cycle?
- Does part of the system need to independently scale?
- Does part of the system need failure isolation?
- Does part of the system need isolation from external components?
- Do you need the freedom to choose the right tech for the job?
- Do your developers waste time waiting for someone else to unblock them?

If you answer "yes" to any of these questions, microservices can be worth the extra complexity.

Once you decide where microservices make sense, how can you adopt this pattern with a bare minimum of complexity? You need the digital scaffolding to make microservices work properly. You will have far greater success with microservices if you run them on a modern, automated platform. *Just ask your peers*.

Platforms such as TAS include *guardrails* to keep services running smoothly on the platform, even as they change constantly. TAS also makes several promises about its behavior. Developers, operators, InfoSec professionals and architects find these promises—and the corresponding predictability—useful.

So take stock of what's important before continuing. What's important is you. Your business, your app idea and your mindshare are baked into a software artifact. Focus and invest here heavily. Invest in your microservices, apps and modern development practices. Leverage frameworks and platforms to handle infrastructure and other undifferentiated lifting. After all, what matters to your business is the rapid delivery of high-quality software that your customers love.

All that said, let's move on to our collection of Spring Boot apps.

Booting up with Spring Boot: A quick start

Baseline apps

Spring has come a long way since *Rod Johnson* created it in 2003. Spring Boot, now in its second generation, provides useful pre-configured Beans for core abstractions such as web servers, security, data-access and connectivity. The addition of Spring Boot was simply the catalyst to make Spring the de facto choice for development on the JVM.

These days, Java developers can build cloud-native apps with a rich ecosystem of tools, *projects* and JVM-based languages. The convention over configuration approach is overwhelmingly popular, and helps enterprise developers get on with the job of writing great software without tiresome, upfront configuration.

About Spring Beans

A Spring Bean—or sometimes just Bean—refers to a piece of code that is managed and created by Spring. For example, you may have a piece of code that manages a shopping cart function or account settings. You may refer to these as your Cart Bean and Account Bean. It's common to hear Spring developers discuss their Spring Beans. Spring Boot can automatically configure your application and create an ApplicationContext with a set of Beans managed by Spring. This opinionated manner of building applications frees developers from hand-configuring things such as servlet containers, data sources and security elements.

Under this backdrop, let's explore some of the finer details of Spring Boot from the application developer's perspective. Non-developers should get something out of this, too, especially when it comes to running these examples in TAS on your chosen cloud infrastructure.

Ok, oil those protractors. Let's get started with a picture of what we're building. The following figure shows our base collection of microservices. As we go, we'll add features to this baseline. For our grand finale, we'll push it all to TAS.

You can see we have a simple front-end UI with a back-end API, represented by these pieces:

- Todo(s) UI implemented in Vue.js (a fork of *todomvc.com*)
- Todo(s) Cloud Gateway in Spring Cloud Gateway
- Todo(s) API implemented in Spring Boot

Let's take a closer look at each app.

Todo(s) Cloud Gateway

Todo(s) Cloud Gateway is a Spring Boot microservice that functions as an API gateway and router for Todo(s) microservices. Based on *Spring Cloud Gateway*, it runs a gateway defined by routes in application.yml by default. When Todo(s) gateway boots into a Spring Cloud environment, it syncs with *service discovery* and loads microservice routes dynamically as they come online. Similarly, it removes microservice routes when they go offline. (We'll discuss Spring Cloud in more detail later.)

Todo(s) UI

Todo(s) UI is a *Vue.js* implementation of *todomvc.com*. We'll use it to interact with *Todo(s) API* directly through the Todo(s) Cloud Gateway. There is a major difference between this version and the one on todomvc.com. Our flavor calls a backing API to create, retrieve, update and delete Todo(s). We perform this action with *VueResource*, which makes HTTP calls to create, read, update and delete Todo(s). (And thus a few log statements are different here compared to the Vue.js app on todomvc.com.)

TAS: A flexible app platform that supports a range of frameworks

Todo(s) UI is simply HTML, JavaScript and CSS. It's not based on the JVM. This showcases another attribute of TAS: the platform supports a wide range of apps. We believe every app can benefit from a modern platform. TAS connects front-end and back-end developers, and enables full-stack software delivery using your favorite frameworks.

Todo(s) API

Todo(s) API shows how easy it is to build API-driven microservices using Spring Boot. If you're just getting started with Spring Boot, then Todo(s) API is a useful way to learn. It implements an API back-end for Todo(s) UI. By default, the API saves Todo(s) in a map that is sized according to the todos.api.limit property. (In the next section, we'll see how to change size limits using Spring Cloud Config Server.)

With @RestController and @RequestMapping annotations on a class, we can encapsulate and provide context for an API. Todo(s) API maps HTTP requests on the root context/to CRUD methods. The Todo(s) Data microservice exposes a similar CRUD API but with zero code from us. Instead, it uses Spring Data Rest to blanket a data model with a CRUD API. Check out *Todo(s) Data* for more information on Spring Boot and how Spring Data Rest can expose your data model as a set of REST endpoints.

API operations

The TodosAPI class contains methods that get called from TodosUI. As you can see, it's very simple to define RequestMappings to handle HTTP requests. Likewise, it's just as easy to define the same RequestMappings as non-blocking endpoints using Spring Boot 2.0's reactive stack.

New to reactive? Welcome to the party.

If you're just now hearing about a reactive stack, you're right on time. While the premises for reactive development patterns have existed for quite some time, we now have practical experience at a new level of scale. Reactive architectures allow applications to act in response to something in an asynchronous manner. This way, creators and users of data can scale independently. You might have used a messaging system at some point in your career. That sounds familiar, right? Now, imagine if your application code was also written with a response-to-events mindset. Well, that would probably be classified as a reactive application.

Spring MVC	Spring WebFlux

`@PostMapping("/")`
`public Todo create(`
`  @RequestBody Todo todo) { }`
 	
`@GetMapping("/")`
`public List<Todo> retrieve() { }`
`@GetMapping("/{id}")`
`public Todo retrieve(`
`  @PathVariable Integer id) { }`
 	
`@PatchMapping("/{id}")`
`public Todo update(`
`  @PathVariable Integer id,`
`  @RequestBody Todo todo) { }`
 	
`@DeleteMapping("/{id}")`
`public void delete(`
`  @PathVariable Integer id) { }`
	`@PostMapping("/")`
	`public Mono<Todo> create(`
	`  @RequestBody Mono<Todo> todo) { }`
	
	`@GetMapping("/")`
	`public Flux<Todo> retrieve() { }`
	`@GetMapping("/{id}")`
	`public Mono<Todo> retrieve(`
	`  @PathVariable Integer id) { }`
	
	`@PatchMapping("/{id}")`
	`public Mono<Todo> update(`
	`  @PathVariable Integer id,`
	`  @RequestBody Mono<Todo> todo) { }`
	
	`@DeleteMapping("/{id}")`
	`public Mono<Todo> delete(`
	`  @PathVariable Integer id) { }`

Now that we've outlined the base parts, let's get to the fun stuff—pushing our app to TAS!

Time to push our code to Tanzu Application Service

Before we blast off, make sure you've installed the *cf-cli* and covered off on the *simple prerequisites*.

The first thing we need to do: Log in to TAS. (Need a quick option? We recommend *Tanzu Web Services*—it's great and has a free trial.)

The only artifacts we need (besides our code) are a manifest and an optional vars file. Both items describe our soon-to-be-live cloud app, and both pay huge dividends on Day 2 when we're running in production. See *this repo* for more information on pushing these apps to TAS.

The vars file

Our vars file contains the dynamic elements of our deployment. It's where we put details about the things we expect to change. For example, we include the public route for each application and the corresponding app artifacts. There's a sublime detail here that developers love: We give our apps to TAS in their native form. Then, TAS bakes and runs them as real-world things. Any developer-centric platform should offer a paper-thin atmospheric layer between source and software, and that's just what TAS is built for.

Here's an example vars file that defines property values specific to our push.

```
api:
    route: https://todos-api.cfapps.io
    artifact: todos-api/target/todos-api-1.0.0.SNAP.jar
ui:
    route: https://todos-ui.cfapps.io
    artifact: todos-ui/

gateway:

    route: https://todos-cloud-gateway.cfapps.io
    artifact: todos-cloud-gateway/target/
    todos-cloud-gateway-1.0.0.SNAP.jar
```

The manifest

A manifest provides a simple definition of our app. These files are easy to control and update via configuration management.

Let's now create a manifest to declare how our 3 applications should be deployed. Variables from the vars file are injected into the placeholders (()). These 2 files are all we need to push. TAS uses the manifest to know how to containerize your application. (This is defined with the buildpack property.) Buildpacks provide the necessary application runtime bits (such as a JVM or an NGinx server) for the container. Want to know more about buildpacks? Check out this *overview* and *our technical docs*. As you may expect, the Java buildpack is a rock-solid way to deliver your JVM-based workloads to TAS.

```
part-1-manifest.yml - app deployment
---
applications:
-name: todos-api
    routes:
    -route: ((api.route))
    path: ((api.artifact))
    buildpack: java_buildpack
-name: todos-cloud-gateway
    routes:
    -route: ((gateway.route))
    path: ((gateway.artifact))
    buildpack: java_buildpack
    env:
        TODOS_UI_ENDPOINT: ((ui.route))
        TODOS_API_ENDPOINT: ((api.route))
```

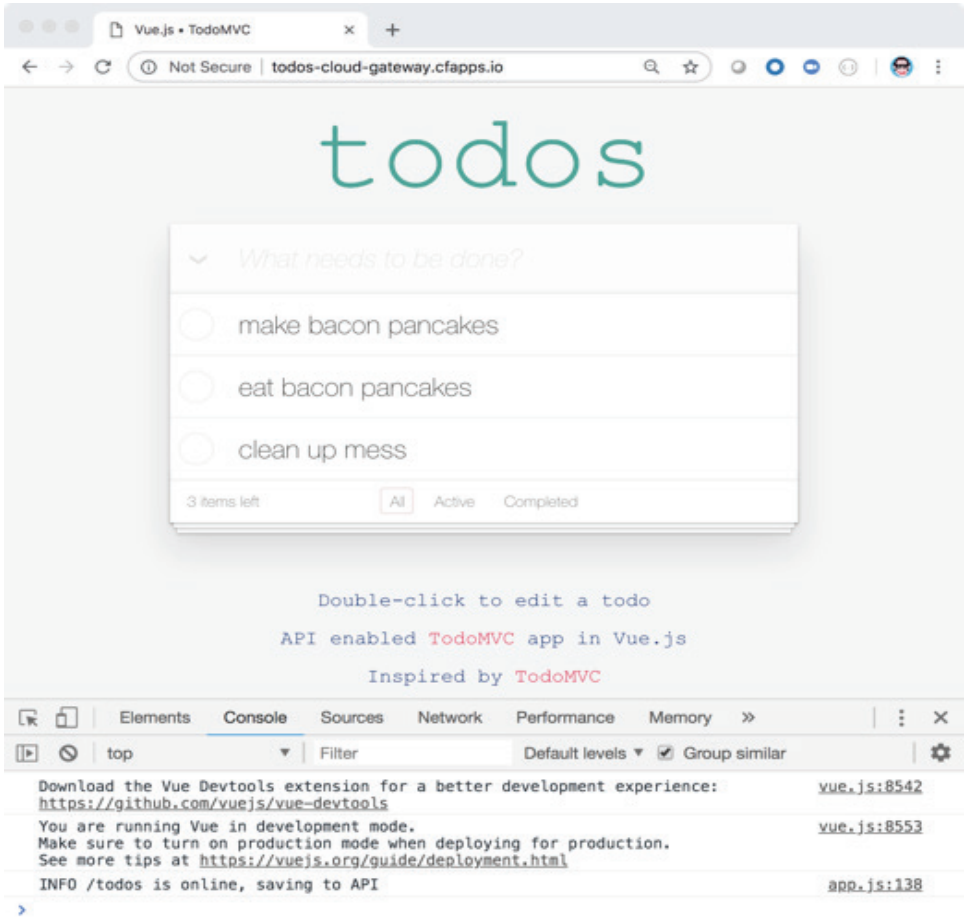
Now, it's time to use the most magical command in cloud: cf push.

```
cf push -f part-1-manifest.yml --vars-file part-1-vars.yml
```

This command launches a task on *TAS to containerize our applications* using the aforementioned buildpack abstraction. TAS takes our application bits, then lays down the appropriate operating system and application stack in a *droplet*. The platform then boots our app into existence. TAS and the cf push command also do many other nifty things for you, such as:

- Create a *secure, hardened container* for you, without unnecessary OS bits
- Setting up a route (or URL) for your application
- Creating a load balancing entry for your application
- Creating SSL termination for your application
- Creating health monitoring and logging subsystems for your application
- Starting your application in a healthy state with the desired number of instances
- Binding any backing services needed to your application by injecting the service credentials

Once the push completes, we can pull up Todo(s) UI in a browser and make a list of really important to-dos.



Take a moment to reflect on what we've just done. We've gone from zero to a full-stack, cloud-deployed application that's darn close to feature complete. (Yes, we'll need to add *single sign-on*. No, we won't do that here.)

"Velocity on the JVM is the killer app"

So we've cobbled a useful Spring Boot app. And we've pushed it to production with relatively little gruntwork. Isn't this what we're after? What we want was *artfully articulated in 2014 by Andy Glover of Netflix Cloud in the SpringOne2GX 2014 keynote*:

Innovation speed in Java **wins**

"Velocity on the JVM is the killer app"

Yes indeed. Velocity on the JVM gives us the capability to quickly release code, learn from it, then release more code based on this feedback. It's a virtuous cycle, and it's the lifeblood of a successful enterprise in the cloud era.

Choose where to run your Java apps, but choose wisely

You have lots of different ways to run a JVM workload. What's going to give us the best velocity? Well, the ideal runtime would:

- Automate a lot of the things you have to do to bring your app online
- Have a simple, consistent and configurable development experience
- Automate containerization
- Provide robust tools and methods for troubleshooting when things go wrong
- Offer a configurable and consistent operational model
- Allow your app to be deployed on different cloud infrastructure environments

That looks a lot like TAS, right? Let's contrast this with other places you can run your JVM-based workloads. Say we have our same baseline set of Todo(s) apps. How would we go about getting the same outcome we just had with TAS?

Consider the numerous ways to start a JVM-based application. You can use various techniques and conventions to pass JVM and application arguments. There's little uniformity and standardization. That makes it harder to go fast.

Let's turn to the Todo(s) UI front-end app. It needs a web-server such as NGinx. In a full-stack environment, a lot of time and effort can be spent configuring and maintaining simple web servers. With TAS, we can easily deploy front-end web apps on forged web-servers, such as NGinx and have routing and load balancing done for us.

What about Docker containers? Sure, folks use it to layer the correct OS and middleware stack together. Once all the apps have the proper Dockerfile, you'd need to build and upload each container image to a container repository. Implementing consistent containerization alone is a huge investment—just ask someone who has tried. What's more, this is a nightmare for InfoSec as there are an overwhelming number of ways to layer containers. You have no idea what's in a given container, or what its vulnerabilities might be. That's why you'll hear folks call hand-built containers "mystery meat." At their best, containers are the output of a scalable, automated process.

Resist the urge to build your own platform—buy a platform and get on with transforming your business

Companies that decide to roll their own platform quickly realize a cold truth: It's more expensive than they thought. In working with numerous Fortune 500 companies, we've found that even a minimal DIY platform effort can take 2 years to build and cost $14 million in payroll alone (for 60 engineers). And that's just to get to a minimum viable product. Can your company wait two years to start your cloud native transformation in earnest?

Of course, once the platform is live in production, the costs continue to grow. As architecture patterns and technologies continue to evolve, the demand for new features can prove challenging to keep up with. Developers may demand that capabilities to support emerging patterns—like the use of a service mesh or functions as a service—be evaluated and added. In addition to any new features, platform operations teams must also manage the lifecycle of the underlying software components and operating system images. They have to quickly take action in response to any vulnerability threats and apply the latest security patches.

Adding these new features to a custom platform and keeping it secure requires a corresponding investment in engineering staff. After a while, your platform team starts to look a lot like a cloud platform software company, with one important difference: The platform you have built doesn't actually generate revenue for your core business. It generates expense and quickly accrues technical debt.

This expense can't be justified, unless your core business is selling cloud platform services. You don't want your most talented technologists building a platform. You want them working on adding unique, differentiating business value. Thankfully, the problem of creating a vibrant, scalable and secure enterprise platform has already been solved by Tanzu Application Service.

Customers such as Boeing, Mastercard, T-Mobile, StubHub, Rabobank, Dick's Sporting Goods, Liberty Mutual and Comcast are all using our cloud native platform to ship high-quality software faster than ever before, enabling their businesses to innovate and thrive as a result.

So of course you can accomplish the same outcomes with Docker, Kubernetes or a homebrew stack you built yourself. But the out-of-box experience of TAS is uniquely simple and elegant. Remember, we don't just want a JVM in the cloud, we want velocity and delivery of great software. TAS provides a great platform experience and allows pattern-oriented extension.

Spring and TAS offer unique developer productivity, especially when compared to other alternatives.

Spring Cloud: A simple way to expand our capabilities

As we've just seen, TAS is a feature rich application platform. As your application deployments increase in size or velocity, TAS provides the infrastructure and a consistent DevOps experience and application environment. For example, when looking at our Spring Boot apps, we see a nice alignment around properties and the environment TAS bakes. Let's say you declare an application property such as todos.api.limit=128. That property can be externally hydrated from the environment TAS creates like so:

```
cf set-env your-app TODOS_API_LIMIT 128.
```

Wonderful stuff, but inevitably, you'll require more than just a consistent environment as your application portfolio grows; for example, cloud primitives such as configuration, discoverability, resiliency and traceability. The goal, of course, is to provide a means for apps and microservices to interact in a digital ecosystem, and that's just what Spring Cloud enables, so you can focus on writing meaningful code, not boilerplate.

Spring Cloud rises to the rescue with handy building blocks such as *config management*, *service discovery*, *circuit protection*, *routing* and *tracing* in a simple-to-reason-about framework. Cloud native development on the JVM has never been so easy and extensible.

For a more detailed look into Spring Cloud consider these resources:

- *Spring Cloud Services for TAS* – Creates and manages Spring Cloud Config Server, service discovery and circuit protection, so you don't have to.
- *Spring Cloud Data Flow for TAS* – Automates provisioning Spring Cloud Data Flow to orchestrate Spring Cloud Streams and Tasks, allowing you to focus more on what the Stream or Task should do and less on how it's being done.
- An expanded Todo(s) sample with Spring Cloud goodness ready to cf push – *This sample set* brings together pieces from the larger Spring ecosystem, including *Spring Cloud Config*, *service discovery*, *circuit protection*, *Spring Cloud Gateway*, *Spring Cloud Sleuth*, *Spring Cloud Streams*, *Spring Data JPA* and *Spring Data Redis*. All essential tools for the cloud native developer.

Running your apps on Tanzu Application Service: What developers need to know

Earlier, we talked about the experience of building an app and pushing it to TAS. Now, let's examine how we can run our apps in the cloud with little to no effort on our part.

Let's continue on with TAS as our deployment target.

Go ahead and log in. Remember, you need the *cf-cli* installed and the *prerequisites* covered off.

So what commands will we need to run our app on TAS? Glad you asked! Let's step through them now.

Apps are first-class citizens

TAS is an application platform. That means the denomination of TAS is an app. This may seem obvious, but it's a key point. Almost every interaction we have with TAS will require us to specify the app along with the relevant operation.

Pushing

Push an application to TAS using vars-file to set variables specific to a push operation. We use vars-file quite a bit with the samples.

```
cf push --vars-file vars.yml
```

Events

This command lists the lifecycle events for an application. You'll use this often, as it's handy to see when, what, who and why an event occurred on your app.

```
cf events todos-api
```

SSH

Hopefully, you don't need to SSH into the container. But when you do need to, it's easy to securely jump into the container running our application.

```
cf ssh todos-api
```

Once SSH'd into the container, it's helpful to know your way around. By default, you land in $HOME for the user vcap (whoami=vcap). Listing $HOME will reveal /app, which is where our application bits live. Because most of our samples are based on Spring Boot, you'll notice a .java-buildpack directory. This contains the whole JVM middleware stack.

Starting, Stopping, Restarting, and Restaging

TAS give you lifecycle control of your apps. You can start, stop, restart and restage each app independently. Starting and restarting are simple commands. Both will apply any new environment variables and services bound to the application.

```
cf start todos-api
```

```
cf restart todos-api
```

Restaging is a repave operation. That means the container will be recreated from the existing application bits and the buildpack. And similar to start and restage, this operation will apply any new environment variables and services bound to the application.

```
cf restage todos-api
```

Buildpacks

As we hinted at earlier, buildpacks provide runtime support for applications. They do this by detecting the type of application, paving the required app framework and integrating your code into a well-formed container. TAS takes your application in its native form as input, and applies the buildpack during the cf push process. The output is a container image that's running on the platform.

TAS comes out of the box with a set of *system buildpacks* ready to automate the source-to-container process for a wide array of languages, such as Java, Kotlin, Groovy, .NET, Node.js, Python and Ruby. Developers have the freedom to use their preferred framework—this is a must-have if you want to go fast at scale. There are lots of choices for developers and streamlined code-to-container automation, which is essential for going fast, especially within a large enterprise. Big companies have thousands of apps. That means you have a lot of variation at the application layer when it comes to frameworks, dependencies and security artifacts. Buildpacks help normalize how these things are containerized. This pays huge dividends for developer productivity and also operationally, especially when it comes to consistency and compliance.

Java buildpack

The Java buildpack containerizes JVM-based applications. It supports a set of *standard container components*, such as:

- Java main()
- Servlet containers: Apache Tomcat
- Application frameworks, such as Spring Boot, Ratpack and Play

Additional application framework components are baked into the container to support a wide range of handy features, such as application performance management (APM), *cloud auto reconfiguration* and *container security*. Last but not least, the Java buildpack comes with standard JREs ready to use as is or with custom settings.

Spring Boot and Java buildpack

As previously described, the Java buildpack provides full container automation for Spring Boot apps, so you can push packaged Spring Boot jars or distZip files. Convenient, huh? There's more. The Java buildpack automatically configures the injection of the Spring cloud profile, if present. In turn, this auto-configures cloud-scoped Beans and configuration.

LET'S SUM IT ALL UP

Buildpacks strike a nice balance between developer freedom and operational consistency. They automate the otherwise hard to build and manage pieces, and abstract away a huge chunk of middleware complexity. Now that's true DevOps enablement.

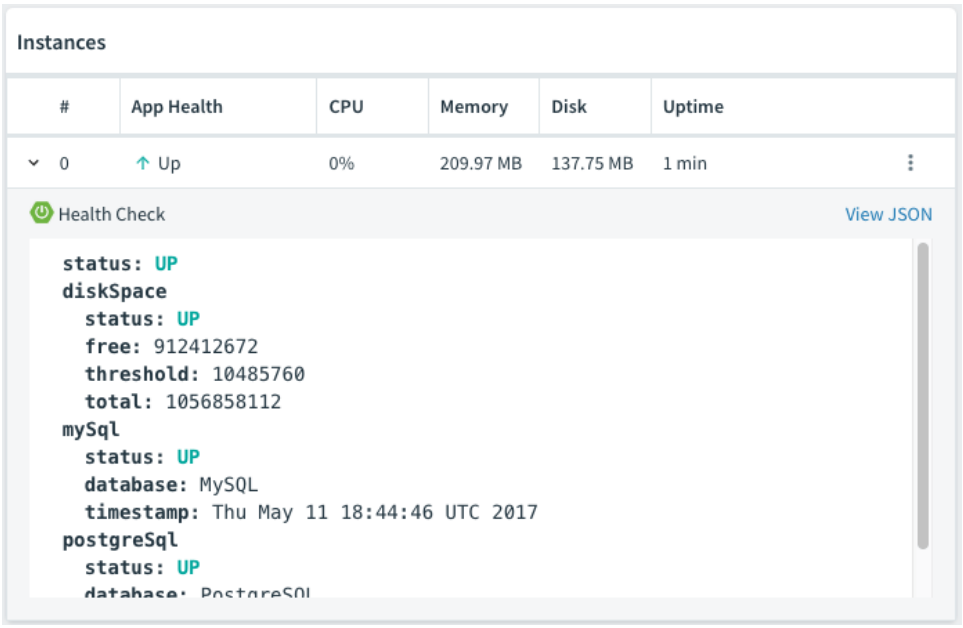
Get visibility into your production apps with Spring Boot Actuators

As VMware advocate *Mark Heckler wrote* ,"Applications are just projects until they're live in production. And once they are live, both developers and operators need the best visibility possible into those critical apps in order to track provenance, monitor health and rapidly troubleshoot indicators of unexpected behavior."

Spring Boot Actuator endpoints are a useful way to gain this information. So Tanzu Application Service integrates several of these endpoints into *its Apps Manager module*. These integrations provide helpful visibility into the state of your apps running in prod. Here's a list of the Actuator endpoints supported in TAS today:

- /info – Exposes details about app environment, git and build.
- /health – Shows health status or detailed health information over a secure connection.
- /loggers – Lists and allows modification of the levels of the loggers in an app.
- /dump – Generates a thread dump.
- /trace – Displays trace information from your app for each of the last 100 HTTP requests.
- /heapdump – Generates a heap dump and provides a compressed file containing the results.
- /mappings – Displays the endpoints an app serves and other related details.

And because a picture is worth a thousand words, here's a screenshot that shows how this data is bubbled up for you.



Quickly troubleshoot cloud-native applications with Tanzu App Metrics

Remember when we said microservices get easier when you have a platform that handles a good chunk of the complexity for you? Tanzu App Metrics, the integrated metrics module for TAS, is one great example of this principle at work. *Here's why*:

"Today's architectures have exponentially more complexity. The question to answer isn't 'what's wrong with my source code?' Instead, teams need to address a series of questions when issues arise:

- Which component of the workload is having a problem?
- How do we trace the relevant requests through the entire workload?
- How do we find all diagnostic information from the components that processed that request?
- And how do we do all of this as soon as possible?

The situation is compounded further when different teams own different pieces of the workload. Why? A select few have an end-to-end understanding of the entire workload.

But just as the industry has rallied around microservices, so too have we rallied to simplify the troubleshooting of these modern, distributed systems."

App Metrics is a useful tool to help you run microservices at scale. You can answer these questions more quickly using App Metrics:

- What went wrong and when? Application metrics provide the answer. Metrics reveal the time window when high network latency occurs and average response times.
- Why did it go wrong? Logs and metrics—such as API calls, exact response times and parameters passed—together usually provide enough information to figure out why latency occurred.
- Who is the culprit behind what went wrong? Events include external bits of information, such as when app source code was changed and whether an app was moved to another container on an underperforming cell. Correlating events with metrics and logs is often the final piece of the puzzle.

Telemetry data for your apps (logs, metrics and events) is rendered visually, along an interactive timeline. Consequently, it's easier to understand sources of latency and system failure. App Metrics also includes distributed tracing capabilities (Trace Explorer) for applications enabled with Spring Coud Sleuth. This helps you understand the sequence of method calls across microservices.

App Metrics has a special bonus for Spring Boot developers. Remember Spring Boot Actuator endpoints from the previous section? App Metrics ingests all Actuator metrics. This information combines with the data already visualized in App Metrics to give you deeper context. You gain even more insight into how your Spring Boot apps are running in production. Use this feature, and reduce the mean time to resolution (MTTR) for your Spring Boot apps.

Effortlessly scale your apps to meet customer demand with App Autoscaler

There's a good chance your applications will experience variable load. How can you scale your app up and down under unpredictable circumstances? Use *App Autoscaler*. This module automates horizontal scaling for your apps. Add more capacity to your applications when traffic spikes. Scale down as traffic returns to normal. Best of all, you can set thresholds and triggers, so the platform does the scaling for you according to rules you configure.

Built-in application logging with Loggregator

Logs are an essential tool when you need to troubleshoot your apps. That's why TAS includes *Loggregator*, a system that gathers and streams logs and metrics. App developers can use Loggregator to "tail" their application logs or dump the recent logs from the Cloud Foundry Command Line Interface (cf CLI), or stream these to a third-party log archive and analysis service.

Keep your apps online serving traffic with four layers of high availability

TAS includes redundancy and resiliency across *four layers of the stack*: monitored processes, health management for app instances, health management for virtual machines (VMs) and availability zones.

In case of failure in any of these layers, TAS automatically recovers to keep serving traffic.

Need a cache, message broker or relational database? Use service brokers.

To do anything interesting with your app, you're going to need backing services. The *Open Service Broker API* gives you a standard set of commands and behaviors that are identical across hundreds of different add-on capabilities. All services provided by TAS implement this API. Thus, you can create and bind a cornucopia of services using the same repeatable motions. Do you need a cache, message-broker, NoSQL store or relational database? You can instantly add them to your app. TAS offers ready-to-use service brokers for *Redis*, *RabbitMQ* and *MySQL*. All of these brokers support an array of server-side use cases, so you can focus on your code and easily add what you need, when you need it.

Hear that sonic boom? That's you going from idea to working software in less than a day.

Time to recap.

We started off the paper seeking answers to a few questions, namely "What's so special about Tanzu Application Service?" and "why does it run Boot apps so well?" Well, by now, you know that TAS:

- Automates a lot of the things you have to do bring your app online
- Has a simple, consistent, and configurable development experience
- Automates containerization
- Offers a configurable and consistent operational model

And you can run your Spring Boot apps on TAS as is, with little or no modification. Plus, TAS plays nicely with the larger Spring ecosystem of projects, so you have an easy way to add more capabilities to your apps over time.

Speaking of apps—just how many apps do you have in your enterprise by now? Hundreds? Thousands? No matter the number of apps, TAS is up to the task. Buildpacks and service brokers give you plenty of flexibility to run all kinds of apps on the platform.

Use Spring Boot and TAS, and you're on your way to supersonic development velocity. Of course, there's always more to be done. Check out our links in *Recommended reading* for more best practices that can help you go even faster!

Recommended reading

Blogs

- *Should that be a Microservice? Keep These Six Factors in Mind*
- *Getting Started with Spring Cloud Services on Tanzu Application Service*
- *Love Spring and Spring Boot? Then, You're Going to Love These Projects, Too*
- *Spring Boot 2.0 goes GA*
- *Opening Doors with Spring Boot 2.0*
- *How App Metrics Helps You Reduce MTTR for Spring Boot Apps (and Save Money too)*
- *TUTORIAL: Using Spring Boot Actuator Integrations With Tanzu Application Service*

Slides

- *Spring on TAS*

Videos

- *Flight of the Flux: A Look at Reactor Execution Model*
- *Full Stack Reactive with React and Spring WebFlux*
- *Getting Super Productive with Spring Tools 4 and Spring Boot 2*
- *Guide to "Reactive" for Spring MVC Developers*
- *Scaling Spring Boot Applications in Real-Time*
- *Spring, the Kotlin and Functional Way*
- *Spring Boot 2.0 Web Applications*
- *Spring Cloud Stream – Developer Recipes, What's New and What's Next?*
- *Springing into Kotlin: How to Make the Magic Even More Magical*
- *Upgrading to Spring Boot 2.0*

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 vmware.com