Adopting Salesforce DX

Get your team ready for the future of DevOps

Contents

Intro

The Platform as a Service (PaaS) landscape has changed dramatically since the launch of Salesforce in 1999. Starting out as a Software as a Service (SaaS) company, but branching into a pioneering PaaS with an emphasis on declarative development and clicks-not-code philosophy, Salesforce is one of the longest-standing PaaS platforms still around today.

Salesforce's philosophy was to make building easy. Software shouldn't be complex to install, set up, or customize. In fact, you shouldn't need to install software at all - it should be available to you at the click of button in your browser. Those traditional in-house development teams, expensively toiling away on complex business logic were obsolete - business analysts, even managers, could build line-of-business applications in a few clicks.

But a lot has changed since those early days. While Salesforce approached the problem from the declarative end of the spectrum, allowing ever greater customization of the platform via a simple user interface, companies like Amazon were approaching from the other direction - providing virtualized hardware that traditional developers could build on using their existing patterns, practices and software stacks.

The result of these competing philosophies wasn't a complete convergence of technologies - Salesforce, and platforms like AWS and Azure, have very different target audiences. While Salesforce was democratizing application development and ushering in the era of the citizen programmer, the other players were strengthening their appeal to the traditional developer.

Unfortunately, as the complexity of software grows, so inevitably does that of the teams building it and the tools required to be successful. With the advent of Apex, and later the acquisition of Heroku, Salesforce acknowledged that only so much can be achieved with the declarative approach. Business requirements can be complicated, and modelling complex domains requires more flexibility than clicks-not-code affords. Traditional development teams weren't dead after all.

As a result of this strategy, Salesforce's marketing and development efforts haven't catered to the traditional developer. Although the platform's ability to support the role has grown substantially, it's clear from the APIs and first-party tools that it's not where the bulk of the R&D dollar has been spent.

By way of example, source-driven development has been a mainstay of traditional platforms for decades, but it's relatively new to the Salesforce ecosystem lexicon, and the fact that source code has to be extracted from the Salesforce org, rather than pushed to an org to define its structure and behaviour, is an artefact of that. So although Salesforce has facilitated business productivity in a way that no other platform can claim, it has under-delivered in this specific area.

That is, until Salesforce DX. The cynical reader could be forgiven for thinking DX is a reactionary move driven by a failure to acknowledge the gradual divergence of the ideals *of No Software* and the practicalities of building and maintaining complex software systems, and to some degree that's true. But it's more than that - it's an acknowledgement that change is necessary, that there are already traditional development teams building business-critical functionality on the platform, and that these teams need to be able to adopt industry-wide practices that developers on other platforms have been benefiting from. It's paving the path trodden by these trailblazers for teams that haven't been able to make that leap yet, transforming a hard-fought expedition through the wilderness into a stroll in the woods, accessible to all technological backgrounds. It's a seismic shift in the philosophy of development on Salesforce.

And it's just the beginning. DX is a toolkit to democratize modern DevOps practices on the Salesforce platform, and today it's undeniably in its infancy. The low-level tools that comprise DX cater to a niche group of very capable early adopters who can piece together command line tools, work through and around any teething troubles that DX might throw up, and feed back to the team at Salesforce. But the DX team is iterating quickly. They're hungry for feedback and picking their battles pragmatically, which is already yielding dramatic improvements like the automated metadata coverage report, splitting up larger, compound objects like custom objects and their translations, and second-generation packaging.

Now we've identified the motivation for Salesforce DX and its high-level purpose, we come to the next question.

Who is this whitepaper for?

This whitepaper is for anybody who wants to know more about Salesforce DX, and how it might be used to implement a DevOps process that works for technically diverse development teams.

So far, we've established that DX caters to the experienced developer with a strong technical pedigree. But these developers don't work in isolation, and we know Salesforce teams are made up of people with different backgrounds and skill sets. No matter where you fall on the scale from no-code to all-code-all-the-time, DX will likely impact you, potentially changing the way you work.

With that in mind, if you're involved in the administration, development, maintenance, or management of Salesforce environments, then this whitepaper is for you. We'll give you an overview of Salesforce DX, an understanding of its motivating factors and the workflows it facilitates, practical considerations for its adoption, and a summary of its pros and cons as things stand today. If you're an admin, architect, developer or team lead considering trialing and adopting DX for your team, then this whitepaper has something for you.

Why Salesforce DX?

Salesforce DX is a means to an end, not an end in its own right. Your goal shouldn't be adoption of Salesforce DX, but to understand how it can help you achieve your development and DevOps process goals. DX is a toolkit that makes it easier for Salesforce developers and teams to adopt better DevOps practices - things like:

- Release automation, including continuous integration
- Automated unit testing
- Automated backup
- Automated rollback, including org change monitoring

The benefits of adopting these practices are myriad and well understood[1] - the result is a happier, more engaged development team with a tighter feedback cycle, meaning fewer bugs, faster fixes, and drastically more regular and reliable releases of new features. This translates into a more effective team and improved business outcomes.

DX isn't a requirement of adopting these practices. As we suggested in the previous section, there are teams who have been following these sorts of practices for years using a combination of existing first and third-party tools. DX is a concerted effort to democratize DevOps, making these practices accessible to more diverse teams by refining existing tools and adding entirely new features.

In the sections that follow, we'll start by talking through the principles of release management, and discuss how the coalescing of these under the DevOps umbrella has changed the way teams handle deployments and automation. We'll then go on to discuss how Salesforce DX fits into this picture and what benefits it affords, how to adopt DX as part of a multi-disciplinary team, and finally we'll touch on some of the current shortcomings and open questions around DX.

What is DevOps?

The meteoric rise of Salesforce has provided us with a platform that delivers on the *No Software* promise. We no longer need to deal with legacy on-premise software with large up-front costs, and instead have a platform that's easy to adopt, and grows with the needs of your business.

There was a mistaken belief that *No Software* meant there would be no need for a solid software development lifecycle. But Salesforce, like all technology that powers our organizations, requires rigorous change management to ensure that changes are effectively communicated from the business to the implementation team, configured and built correctly, tested and verified to ensure it delivers what the business originally wanted, and finally deployed to production in a way that bears benefit to the end users.

"Release management" is a popular term used to describe the process of planning,

---

[1] https://puppet.com/resources/whitepaper/state-of-devops-report

building and releasing software on Salesforce. But with the advent of DevOps, there's been a subtle shift in terminology used to discuss the operational issues of releasing software on the Salesforce platform. The term "DevOps", although prevalent on other platforms for the last several years, has only recently fallen into common parlance in the Salesforce development community.

These terms aren't synonymous. Release management encompasses a broader set of activities than DevOps. As the name implies, DevOps refers to the technical, operational aspects of releasing software - packaging, releasing and monitoring changes through the software development lifecycle. Release management layers the less technical aspects of planning and managing releases on top of this.

Beyond that, DevOps also has implications of responsibility. Rather than a separate release manager or team, DevOps comes with the implication that the responsibility for packaging, releasing and monitoring is shared across the whole development team. It also implies a heavy reliance on automation, allowing teams to release regularly and reliably with confidence, and much shorter release cycles as a result. With these process improvements, there are fewer bottlenecks and fewer barriers to releasing software into production, fewer defects, and better business outcomes.

The non-operational aspects of release management are well understood, and more or less equivalent from platform to platform. Crucially, tools to facilitate this part of the process also span platform boundaries - there are many planning, management and collaboration tools that have been refined over many years to cater to teams of all shapes and sizes. Teams across all software stacks and platforms are well-served by these tools. The operational aspects, however - those that fall under the DevOps banner - are closely bound to their associated technologies.

DevOps and Salesforce

Salesforce teams are behind the industry curve when it comes to DevOps. There are a variety of opinions as to why[2], but the biggest factor is that one of Salesforce's strongest selling points and early USPs was that traditional developers and ops specialists were unnecessary - the ethos of *No Software*.

Salesforce's entire approach was different to that of traditional technology stacks and the cloud ecosystem that evolved around them, and so the tools to adopt similar practices on top of Salesforce didn't really exist. Moreover, the decision to adopt Salesforce is more strategic than any other cloud provider or tech platform - it's a business decision and very scarcely technically-lead, so developers are rarely involved until later in the process. Traditional enterprise developers haven't been the target audience, and Salesforce hasn't needed to court them as closely.

It may seem strange then that the trend is towards DevOps. But DevOps for Salesforce doesn't look quite the same as on other platforms, and this helps explain its increasing popularity. Firstly, the well-known Salesforce platform benefits of

---

[2] http://ivanesalmeida.blogspot.com/2018/07/salesforce-and-devops-part-1-my-views.html

agility, speed of execution, and not needing large teams of developers still hold true - these only start to become strained when projects grow dramatically in complexity. Secondly, Salesforce itself removes a lot of the complexity you might find in DevOps on other platforms. Managing infrastructure, scalability, hosting, even tests - traditionally the responsibility of ops personnel and their chosen systems and tools - is all handled by the platform itself. DevOps on Salesforce is a narrow subset of DevOps on other platforms.

So what's left within the remit of DevOps in Salesforce? As we mentioned earlier[3], there are a few key areas:

- Release automation, including continuous integration
- Automated unit testing
- Automated backup
- Automated rollback, including org change monitoring

Again, DX doesn't directly perform any of these tasks - instead, it provides some building blocks to help you put together these workflows yourself. By way of example, an automated release process might consist of the following:

- A scheduling / automation server, like TeamCity or Jenkins
- Your source control provider, like GitHub, GitLab or Bitbucket
- Your Salesforce environments - DE orgs, sandboxes, prod, and scratch orgs
- Tools like the Force.com migration tool or the Salesforce CLI for talking to Salesforce's APIs, alongside the git CLI for pushing to and pulling from your source control system
- A series of bespoke scripts that pull these disparate tools together, designed to perform deployment-related tasks like packaging and pushing metadata, or running unit tests

You'll notice that although DX makes it easier to build this sort of solution, it's always been possible to build a workflow like this with existing tools. There are several benefits DX offers over the legacy tooling, but first, let's take a look at the main DX features.

Salesforce DX - an overview

DX consists of several key features:

Scratch orgs

Scratch orgs[4] are ephemeral orgs. They can be configured in terms of edition, feature enablement, and preferences with a JSON file (you can, for example, create a scratch org with person accounts enabled without having to contact support), and created with the Salesforce CLI. Once created, metadata and code can be pushed to them and pulled from them with another CLI call. They're ephemeral - they'll be deleted after a

---

[3] Why Salesforce DX

[4] https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_scratch_orgs.htm

maximum of 30 days, but their durations can be configured at creation.

Scratch orgs are designed to be short-lived and used for day-to-day development. Their main purpose in a DX-based process is to replace dev sandboxes. The process is as follows:

- When picking up a new work item, create a new branch in your source control repo, and a new scratch org, and push your metadata and code from your branch
- Make changes to the scratch org, modifying metadata and code, regularly syncing your changes back to your branch in version control
- When the work item is complete, delete the scratch org and merge the feature branch into the master (or equivalent) branch for the repository

This approach gives you a clean starting point each time you start work on a new feature or fix. Scratch orgs can also be used for:

- Testing a feature: as a test engineer, grab a feature branch, create a new scratch org, and push the source to the scratch org
- Automated testing: on commit to a branch, create a scratch org and deploy the source to it, running all unit tests and reporting on failure

A new metadata format

For those familiar with Ant and the Force.com migration tool, the APIs used by Salesforce DX are the same. Anybody who's used the Force.com migration tool will know that the structure of metadata returned by those APIs isn't ideal for working from version control. Items like custom objects and profiles can get really big, which poses a few different problems:

- Working with large XML files is difficult, just by virtue of their size
- Multiple users changing the same files makes conflicts more likely, and failing to resolve these effectively can yield corrupt XML
- Although not best-practice, it's not uncommon for these items to contain a variety of different changes according to different features in active development in a particular org - this means when pushing changes, developers need to edit these XML files by hand to incorporate only the specific changes they want to push in that deployment or commit

To address this, DX uses a slightly modified folder structure[5] that breaks up some of the larger, more complex metadata items (for example custom objects and translations) into subcomponents, providing a good workaround to these problems. It's worth noting, however, that because it uses the same underlying APIs there are still problems this mechanism doesn't address. It also comes with the added benefit of loosening some of the restrictions on folder structure, allowing you to introduce subfolders to organise your Apex classes into, for instance.

---

[5] https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_source_file_format.htm

The Salesforce CLI

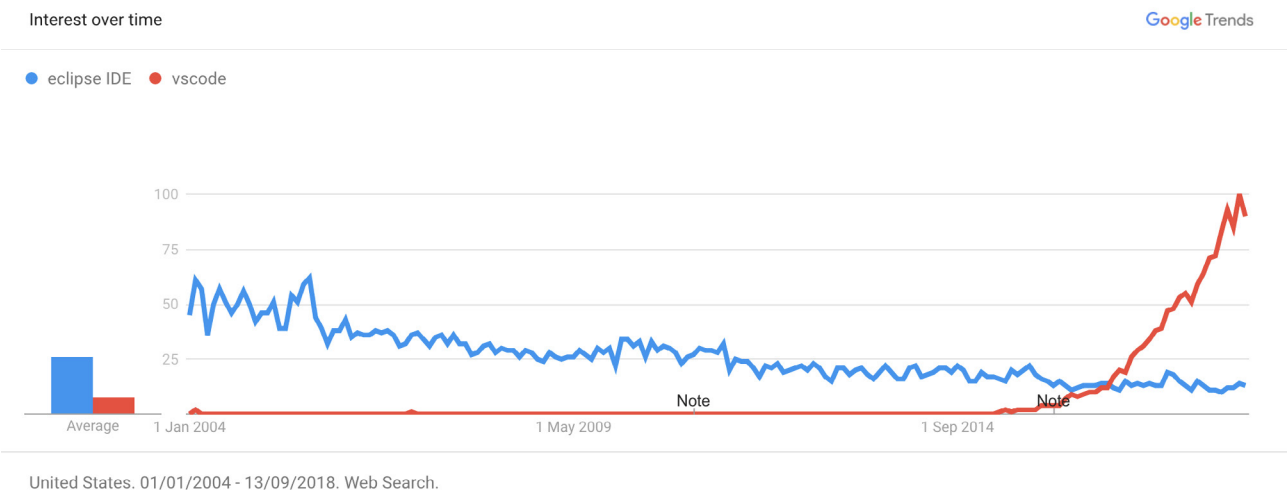
The Salesforce CLI[6] is essentially an iteration on the Force.com migration tool. It incorporates:

- Everything that already existed in the Force.com migration tool, particularly around retrieving and pushing metadata from and to an org
- Commands for translating Force.com migration tool-format metadata to DX-format metadata and vice versa
- Commands for managing the features of DX, specifically scratch orgs and second-generation packages
- Commands for importing and exporting data from and to JSON files
- A handful of other administrative commands to conveniently perform tasks programmatically, including running unit tests

Using the Salesforce CLI instead of the Force.com migration tool allows you to control the new DX-introduced features like scratch orgs and second-generation packaging, and lets you use the new DX-introduced metadata format discussed in the previous section.

Visual Studio Code plugins

Salesforce has announced that the Force.com IDE - a set of tools to bring Salesforce support to the Eclipse IDE - has effectively been deprecated[7], and future first-party development effort in this area will follow popular opinion and target Visual Studio Code[8]. Eclipse is a fine, open-source IDE, and a few years back was the obvious choice to extend with support for new languages. While it's still popular amongst certain demographics (Java developers, for instance), the rise of html-based desktop IDEs over the past few years has caused a fairly dramatic drop in its popularity:



[6] https://releasenotes.docs.salesforce.com/en-us/winter18/release-notes/rn_sfdx_cli.htm
[7] https://developer.salesforce.com/blogs/2018/02/salesforce-extensions-vs-code.html
[8] https://releasenotes.docs.salesforce.com/en-us/winter18/release-notes/rn_sfdx_vscode.htm

Salesforce has followed the popular trend in shifting their development effort to the lighter-weight and increasingly popular VSCode. VSCode's powerful plugin model means Salesforce has been able to put together a full-featured IDE in a short period of time. This model also leaves it open to further extensibility by the community. If you're adopting a new IDE for Apex development, then VSCode should be your first port of call.

Second-generation packaging

DX introduces a new packaging model[9]. Various terms are used throughout the online literature - although there's some nuance, packaging 2.0, second-generation packaging (2GP), unlocked packages and developer-controlled packages (DCP) are all used broadly interchangeably to refer to the same concept. For convenience, we'll stick to "second-generation packaging" to refer to the overall feature, and "unlocked packages" to refer to the packages themselves.

Unlocked packages are designed to solve two main problems:

1. Many orgs' metadata is organized as a "happy soup" - the org is essentially a collection of metadata without structure. Unlocked packages provide a mechanism to encapsulate independent pieces of functionality, making the overall structure of an org easier to reason about and understand, and more importantly easier to deploy. Encapsulation, separation of concerns and single responsibility have long been tenets of good software development practice, and unlocked packages facilitate this.
2. Rather than deploying metadata between environments as a zip file with a package.xml, unlocked packages are designed to be individually deployable units of functionality, built from source control, and deployable to your org. They're versioned, meaning you can track which org has which version of which feature and you can express dependencies on other unlocked packages in a package's definition. Once a package version has been tested in one environment, deploying to a downstream environment is as easy as telling Salesforce to deploy that same package to another org. This has the result of being a more repeatable and reliable mode of deployment.

The result of this is that much more functionality will be delivered by package, and it'll become more rare to deploy raw metadata between environments. You'll add features to orgs by installing packages, and when building out new features or extending existing ones, you'll convert these to versioned packages and deploy them to downstream environments.

Once an unlocked package is installed in an org, its metadata acts as though it belongs to that org and can be modified or deleted like any other metadata, with the caveat that any changes made to that package's metadata in the org will be overwritten if a new version is installed. This discourages teams from making substantial or long-standing changes in production, preferring making changes and fixing bugs

[9] https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_dev2gp.htm

in the package itself, creating a new version, and pushing that downstream via an established release process.

Note that at time of writing, this feature is in beta. It can be risky to adopt beta features aggressively, because there's still a chance they won't make general availability, but it's stabilized a lot over the past few months and we don't see this feature disappearing any time soon. We'll discuss the practicalities of adoption of second-generation packaging in a later section.

Dev Hub orgs

Every scratch org needs a home, and that's where Dev Hubs come in. A Dev Hub is just a standard org that can act as a container for scratch orgs. In order to create a scratch org, you'll first need to convert one of your existing orgs to a Dev Hub[10], or sign up for a Dev Hub trial org[11]. Once you have a Dev Hub org, you can use the Salesforce CLI to start spinning up scratch orgs, and you can view and manage any existing scratch orgs you've created via the Dev Hub org's UI. The recommendation today is to stick to only one Dev Hub org, and usually production.

Dependency API

With Summer '18 came the pilot of a new dependency API[12]. This is a new feature of the tooling API, and exposes relationships between metadata components. With this API, you can write SOQL queries that not only list metadata components referenced by a specific component, but also go in the other direction and list which components reference a specific component. Not all metadata types are covered by this API yet, its output config options are limited to plaintext and JSON, and it's still in pilot, but it's already showing a lot of promise. Although Gearset has its own dependency tracking features, this API will be particularly useful for third parties building developer tooling for Salesforce devs and admins.

DX for Salesforce DevOps

So if you're looking to build a DevOps process around DX, or you're looking to transition your existing process to DX, what's the best place to start?

The lure of "best practice"

When we speak with Salesforce teams, especially those struggling with increasing complexity, there's often a desire to adopt a "best practice" approach to DevOps and their software development lifecycle. We think this is a mistake. There are really three key elements to adopting DevOps on your team:

- Identifying problems you want to solve

---

[10] https://developer.salesforce.com/docs/atlas.en-us.sfdx_setup.meta/sfdx_setup/sfdx_setup_enable_devhub.htm

[11] https://developer.salesforce.com/promotions/orgs/dx-signup

[12] https://www.youtube.com/watch?v=zsZDEL6oO0Q&feature=youtu.be&t=13m1s

- Version control as source of truth
- Adhering to the process across the whole team

It may sound obvious, but if you're looking to adopt or further DevOps in your business, the best starting point is to identify the problems you'd like to fix with your existing process - only then can you design a plan to address them. Perhaps your developers are treading on each other's toes by working in the same environment, or changes are being made directly in production then getting overwritten. Perhaps your deployments are taking days to complete, or you're conducting them out of hours due to their fragility. Whatever the problems may be, DevOps is the remit and responsibility of teams themselves, so charge your admins, developers, business analysts, and release managers with identifying the challenges they're facing.

We think there's a spectrum of best practice and you should implement the parts that are going to deliver the best immediate return on investment, while still laying a foundation to build upon. Start with the problems causing you the most acute pains, and incrementally address subsequent problems in turn. In building Gearset, we've adopted some cutting-edge practices that allow us to release a new version of the app in just a click, and we do this multiple times a day. We didn't start out this way - when we encountered a problem that our process didn't handle well, we refined our tools. In our experience, teams who take this approach, rather than an all-or-nothing approach, are less likely to end up in the "nothing" camp twelve months down the line.

The DevOps maturity matrix

To help guide decision making around DevOps adoption, we use a maturity model to help understand where teams are right now and where they stand to benefit the most:

	Immature	Transitioning	Mature	Trailblazing
Challenges	Disaster recovery impossible, little or no audit trail, time consuming / error-prone manual deployments	Disaster recovery very challenging, little audit trail, some automation around deployments but can be brittle depending on tooling	Environment creation, branch switching, getting whole team on one process, metadata complexity, metadata versioning depending on tooling	Metadata versioning, getting whole team on one process, depending on tooling
Developer org management	Single development org or no development org	Shared single development org	Individual sandboxes	Scratch orgs
Deployment technology	Change sets	Gearset/Ant/Eclipse, etc.	Gearset with CI/Ant + CI server	Gearset with CI using SFDX/SFDX + CI server
Source control	None, or used as backup	Used for development, but limited metadata types	Source of truth for development, with feature branches	Multiple SFDX projects, using unlocked packages
Rollback	None	Manual via pre-defined checklist	Automated using Gearset	Automated using Gearset

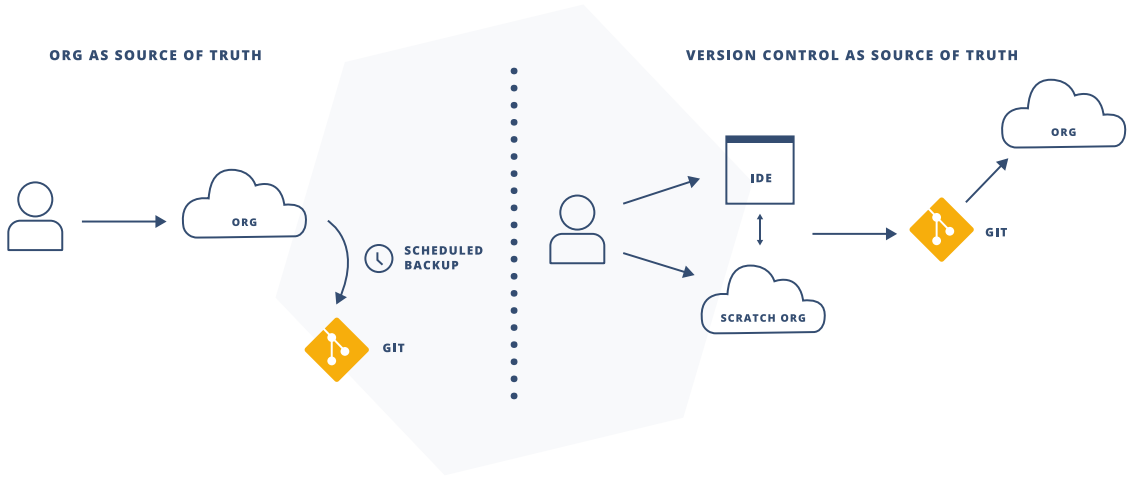
You can identify your current level of maturity by looking at the sorts of challenges faced by teams at each stage, as well as the sorts of tools they're using. Once you've identified the stage of maturity and curated a list of pressing challenges you'd like to resolve, you can put the tools and processes in place to tackle those problems and progress to the next stage of maturity.

The stand-out characteristic here is the relationship between maturity, and segregation of concerns and automation. Generally speaking, teams with more mature DevOps processes will have more environments, keeping ongoing development work more segregated, and this is enabled by a heavier reliance on automation.

Version control as the source of truth

The single most impactful change that Salesforce teams can make on their way to implementing a solid DevOps workflow is to adopt source control as the source of truth[13].

First, it's worth defining "source of truth" in this context. It means that if you discover a discrepancy between your org and the metadata in source control, you trust source control and update your org to match it. It means that if your org were to disappear tomorrow, you could spin up a new org based solely on what you have in source control. This can be a conceptual leap for teams that don't use source control at all, and arguably so too for teams that currently use git only for metadata backup.



From git as backup to a source-driven development workflow

Adopting source control is too broad a topic to cover completely in this whitepaper. If you're yet to make the leap to adopting source control, take a look at our "Version control for Salesforce" whitepaper[14] for a detailed exploration of the topic. Regardless of which source control system you adopt, or how your team is planning to take

[13] https://trailhead.salesforce.com/en/modules/sfdx_dev_model/units/sfdx_dev_model_neworganization
[14] https://gearset.com/assets/version-control-for-salesforce-whitepaper.pdf

advantage of it, the move to using source control as the source of truth is the most impactful change you can make in adopting Salesforce DX and a DevOps mindset.

Salesforce DX for the whole team

In this whitepaper we've explored what Salesforce DX has to offer and all the power that it's unlocking for teams, but we think Salesforce made a rare marketing misstep when it referred to it as the Salesforce Developer Experience. It isn't.

It should be positioned as the Salesforce Builder Experience, as effective adoption and implementation of a DevOps strategy requires full involvement from the whole team. It comes back to the "source of truth" concept that we've talked about throughout - the benefits of source-driven development are only unlocked when everybody works from a single source of truth.

If critical team members like business analysts and admins are excluded from the process then you'll face inevitable friction between the process that they're following and the process that others on the team are using.

Most guides to Salesforce DX are written from the point of view of developers - specifically those that are comfortable using CLI tooling. The challenge with viewing the problem exclusively through that lens is that the learning curve for a CLI-based workflow is steep, and it dramatically erodes the clicks-not-code change velocity that made Salesforce beloved by businesses in the first place.

The good news is that Salesforce has built DX as an enabling platform and made it simple for third-party DevOps leaders like Gearset to build atop the concepts and APIs to provide seamless source driven development for the whole team. In fact, Gearset exposes a full suite of DX features that developers are benefitting from in a familiar clicks-not-code UI, allowing low and no-code developers and admins to follow an identical process to developers on their teams, without the learning curve. All of this has been made possible by the API and CLI-based approach taken by the DX team.

An example DX-based DevOps process

Let's walk through what DX-based DevOps might look like for a team falling under "Trailblazing" in the maturity matrix.

Source control

The team will be using git, the industry de facto standard, for source controlling their metadata. They'll be using a repository hosted by one of the main providers, usually either GitHub[15], Bitbucket[16] or GitLab[17], though it's possible they'll be using an on-premise flavor, hosting elsewhere, or self-hosting.

---

[15] https://github.com
[16] https://bitbucket.org/product
[17] https://gitlab.com

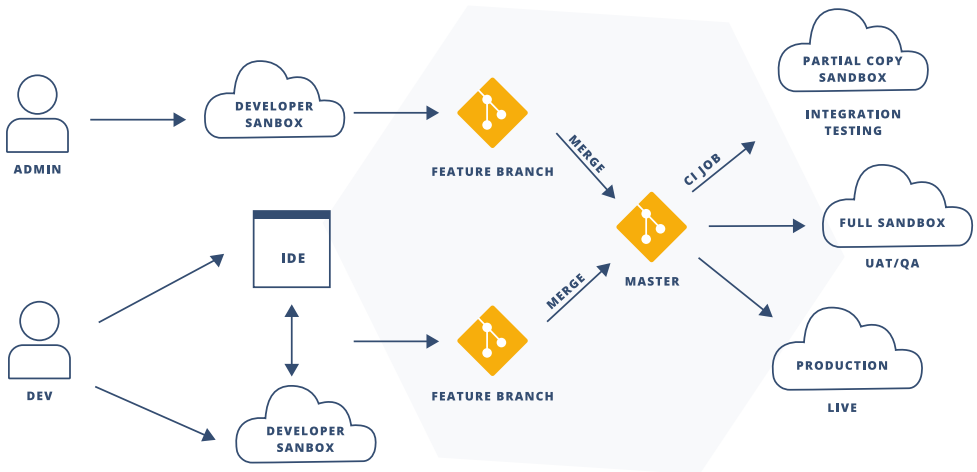
Development workflow

There will be a clear source control-based workflow for making changes to the production org that all team members, whether developer, architect, admin, or business analyst will adhere to. This will usually be based on the popular Gitflow[18] model, although in most cases we see users adopting a simplified version of this model.

A source-driven flow

At a high level, the process is as follows:

- All relevant metadata is in source control - this will start out as a small subset centered around custom objects and Apex classes, but will grow as the team becomes more comfortable with the process and, crucially, as the release cadence increases
- When an admin or a developer commences work on a feature, they'll first create a branch from the master branch of the git repo
- The team should also maintain a scratch org definition file within the git repo, so that developers and admins can spin up a scratch org of a specific configuration, and push the source-controlled metadata to it on commencing feature work as necessary
- The developer or admin then makes changes to scratch org and metadata on disk, and keeps them in sync using their preferred tooling
- When the feature or change is complete, the developer or admin creates a pull request, then assigns this request for review to a colleague
- When the change has been reviewed and any necessary modifications made, the branch should be merged back to master, signifying that the feature is ready for release



A source-driven workflow for admins and developers

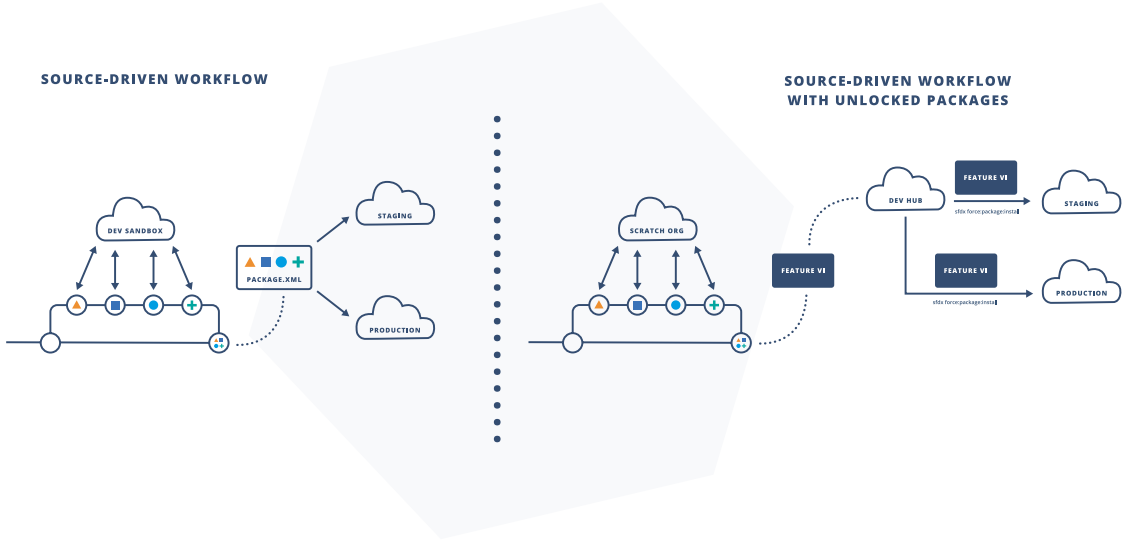
[18] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

To make this process as effective as possible, it's important to break feature work up into the smallest possible deliverable slices. Releasing small features regularly has myriad benefits:

- It increases the robustness and reliability of the release process by exercising it regularly
- The "ceremony" in releasing is reduced - it becomes commonplace
- Smaller changes are necessarily easier to release - the likelihood of hitting an unintended edge case is dramatically reduced
- The likelihood of challenging merges caused by developers working in the same areas of the codebase are reduced

Unlocked packages

This workflow can be modified slightly to take advantage of the new packaging features in pilot in Summer '18. As we've already discussed[19], unlocked packages let you bundle and deploy features and parts of your application much in the same way you would if you were releasing it on the AppExchange. Rather than have a single repository containing the metadata representation of your whole org, and deploying changes by pushing metadata from your git repo directly to your org, you instead create a series of packages representing features of your org, and push these as versioned packages. Once a versioned package is created, you can install this into your org in much the same way you would an AppExchange package.

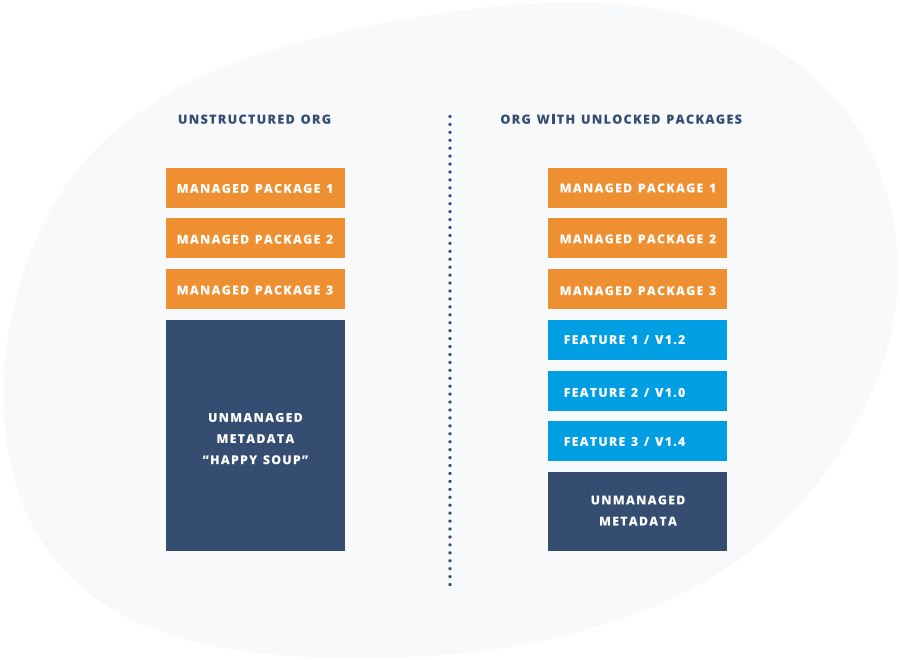


Adding packaging to a source-driven development workflow

Rather than orgs consisting of large quantities of metadata, under this model they're composed from a variety of packages. A package might be shared across several orgs or projects, or it could be a feature specific to a particular org. There are several benefits to this approach:

[19] Second-generation packaging

- Composition is a long-standing best practice of software development, and separating features and functionality into different packages forces better software design practices
- It's easy to understand at a glance the state of an org by looking at which versions of which packages are installed - this provides a high-level description of what functionality is available in an org, without having to look at the metadata itself
- Installing and upgrading features is easy - there's no longer a need to consider which individual metadata items need to be deployed, just tell Salesforce to install the package and all the constituent components will deployed as a single unit
- The risk to deploying a feature is reduced - installing a package is more reliable than pushing raw metadata



From metadata happy soup to unlocked packages

However, this is no panacea. Along with the new packaging approach come far-reaching software design implications. Although this encourages best practices, to execute this approach properly requires strong architectural decision making, and liberal use of enterprise patterns. When relying on packages, teams must carefully consider where responsibilities lie between packages, and perhaps more importantly, what the boundaries between packages look like. Minimising coupling between units of code is a long-standing software development problem, but this is made more challenging by the relative independence and isolation of different unlocked packages.

Moreover, cross-cutting concerns can be difficult - changes that need to be made across several packages have to be carefully made depending on the degree of isolation between packages, and refactoring tools, the likes of which developers

on some other platforms can benefit from, aren't available yet. Decomposing existing large orgs into packages can be very difficult too - metadata can be heavily interdependent and teasing apart these dependencies is challenging.

Ultimately, many of the problems of unlocked packages are similar to those of deconstructing traditional SaaS applications into a microservices architecture - application-spanning changes are difficult, communication boundaries must be clearly defined, and communication channels must be abstract and flexible. As a result, teams adopting packages require very strong and seasoned technical leads, and a well-defined plan for integrating changes made by less technically experienced team members into the development process. Our advice is to approach unlocked packages with caution at this early stage.

Given this feature is still in pilot, adoption is currently fairly limited and the functionality is still shifting over time, though it's started to settle more recently. On the upside, it's yet another opportunity for incremental adoption - if you're interested in using unlocked packages, our advice is to start small with greenfield features or projects, rather than attempting to adopt wholesale across your existing orgs.

Tools

There are a variety of tools your team might use to accomplish this. For the more technical developers, the following would be common:

- VSCode
- The git CLI / GitHub desktop client / SourceTree / GitExtensions / GitKraken
- The Salesforce CLI
- Scratch orgs

For the less technical developers, there are fewer options. It's certainly possible for developers of all technical experience levels to learn to use the above tools, though the learning curve is notoriously steep for those inexperienced with command line tooling. The other option is to adopt a tool like Gearset, which allows users to work with scratch orgs, live orgs, and git repos via a graphical UI.

Release workflow and release automation

Once code has been merged, this will be released to various environments by an automated or semi-automated process. Firstly, there will be a number of permanent environments that might include staging, UAT, and production. Changes to master will be continuously deployed to staging. Once changes have been verified in staging, it will be possible to deploy those changes to downstream environments like UAT and production in just a few clicks.

Tools

There are a couple of different ways to achieve this:

1. An off-the-shelf or open source continuous integration server like Jenkins, in conjunction with the git CLI, the Salesforce CLI, and custom scripts written by the development team to handle common deployment tasks.
2. An all-in solution like Gearset that handles interfacing with git, running continuous integration jobs, and deploying metadata to orgs and scratch orgs

The temptation here might be to reach for the flexibility of an off-the-shelf CI server, but these are generally far more configurable than is required for Salesforce teams. CI servers are designed to be sufficiently flexible to run a variety of different types of job, most of which aren't necessary when Salesforce is the target platform, and this additional configurability comes with a burden of responsibility. Take, for instance, a popular example that uses a combination of the git CLI, the Salesforce CLI and an off-the-shelf CI solution, by Salesforce MVP Daniel Stange[20]. The configuration file for the CI job is as follows:

```
version: 2
jobs:
    setup-dx-environment:
        machine: true
        working_directory: ~/ci_app
        environment:
          # from https://developer.salesforce.com/docs/atlas.en-us.sfdx_setup.meta/sfdx_setup/sfdx_setup_install_cli_standalon
          # and https://developer.salesforce.com/media/salesforce-cli/manifest.json
          - DX_CLI_URL: https://developer.salesforce.com/media/salesforce-cli/sfdx-linux-amd64.tar.xz
        steps:
        - run:
            name: Download and Install CLI
            command: |
                cd ~/
                mkdir sfdx
                wget -qO- $DX_CLI_URL | tar xJ -C sfdx --strip-components 1
                ./sfdx/install
                sfdx
        - run:
            name: create tmp dir and server key; check server key
            command: |
                mkdir ~/tmp
                mkdir ~/tmp/sfdx-keys
                echo $HUB_KEYKEY | xxd -r -ps >> ~/tmp/sfdx-keys/server.key;
                openssl rsa -in ~/tmp/sfdx-keys/server.key -check -noout

        - run:
            name: Authenticate Dev Hub
            command: |
                sfdx force:auth:jwt:grant --clientid $HUB_CONSUMERKEY --jwtkeyfile ~/tmp/sfdx-keys/server.key --username $HUB_
                sfdx force:auth:jwt:grant --clientid $DEPLOY_CONSUMERKEY --jwtkeyfile ~/tmp/sfdx-keys/server.key --username $D
        - save_cache:
            key: sfdx-$CIRCLE_BUILD_NUM
            paths:
                - ~/.sfdx
        - persist_to_workspace:
            root: ~/
            paths:
                - .sfdx/*
                - sfdx/*
                - tmp/*
        - store_artifacts:
            path: ~/.sfdx/sfdx.log
            destination: sfdx-logs

    tests:
        machine: true
        steps:
            - checkout
            - attach_workspace:
                at: ~/
            - restore_cache:
                  keys:
                      - sfdx-$CIRCLE_BUILD_NUM
            - run:
                name: Update PATH and Define Environment Variable at Runtime
                command: |
                    echo 'export PATH=/home/circleci/sfdx/bin:$PATH' >> $BASH_ENV
            - run:
                name: Create a fresh scratch org
                command: sfdx force:org:create -f config/project-scratch-def.json -a testOrg -s
            - run:
                name: Push source and run tests
                command: |
                    sfdx force:source:push -u testOrg
                    sfdx force:apex:test:run -u testOrg -l RunLocalTests -r human -w 10 -c
            - run:
                name: Teardown
                command: sfdx force:org:delete -p -u testOrg
    deploy:
        machine: true
        steps:
            - checkout
            - attach_workspace:
                at: ~/
            - restore_cache:
                  keys:
                      - sfdx-$CIRCLE_BUILD_NUM
            - run:
                name: Update PATH and Define Environment Variable at Runtime
                command: |
                    echo 'export PATH=/home/circleci/sfdx/bin:$PATH' >> $BASH_ENV
            - run:
                name: Convert SFDX source to Metadata Bundle
                command: sfdx force:source:convert -d src-$CIRCLE_BUILD_NUM
            - run:
                name: Deploy Metadata Bundle to target org
                command: |
                    sfdx force:mdapi:deploy -u deploytarget -d src-$CIRCLE_BUILD_NUM -l RunLocalTests -w 10

workflows:
  version: 2
  run_build:
    jobs:
      - setup-dx-environment
      - tests:
          requires:
              - setup-dx-environment
      - deploy:
          filters:
            branches:
              only: master
          requires:
```

There's a lot of complexity in this config, and any team managing this sort of process on their own will need to understand and modify these sorts of configurations

[20] https://github.com/dstdia/ForceAcademy18/blob/master/.circleci/config.yml

semi-regularly, on top of administering their automation solution, version control system, updating scripts to accommodate changes in the DX CLI, and so on. Using a Salesforce-focused solution like Gearset has the benefit that this complexity is handled for you.

Testing

Because of test coverage requirements, and the possibility of breaking unit tests by making changes directly in production, the team will automatically run unit tests across environments on a fixed cadence. Tests will run when pushing changes from environment to environment regardless, but depending on your release cadence it's possible for tests to begin to silently fail, and for these failures to first be discovered at deploy time. Discovering test failures late in the release process can be costly.

By running tests on a regular basis, either with a scheduled job configured on your CI server running tests via bespoke script, or with a tool like Gearset, and notifying the team on test failure, errors can be caught closer to implementation time, which makes them easier to fix.

Monitoring

Particularly while a new process is being put in place, old habits may dictate that changes are occasionally made directly in production. This subverts the ideology of "version control as source of truth", and causes developers to tread on each other's toes and changes to be lost. As such, the team will monitor their production environment for unexpected changes. On detection of unexpected changes, team members can then identify the cause of the change and the person responsible, and decide whether to revert the change or roll the change back into source control. At present the best way to do this is using Gearset.

Backup

The final piece of the puzzle is maintaining a full history of the state of the org, both for audit and for disaster recovery. Source control itself solves this problem, and tools like Gearset bring the added benefit of automated and selective rollback.

Challenges

As things stand today, DX goes a long way towards delivering the tools you need to build a cutting-edge DevOps process. It's not without its shortcomings, though.

Firstly, there can be a steep learning curve for even technically experienced teams. There are lots of moving parts with a variety of diverse tools, and anybody unfamiliar with CLI tools may find themselves out of their depth. That said, it's not insurmountable, and there are third-party tools available that circumnavigate this issue.

With a steep learning curve and diverse tools comes a certain degree of overhead.

Piecing together CLI tools, maintaining continuous integration configurations and servers, and the writing and curation of bespoke scripts to perform tasks like running tests and deploying changes requires expertise and time. While DX is still under active development with several features still in pilot, the task of maintaining this is likely to be more onerous as the commands, APIs, and features evolve over time. At its core, it becomes an issue of prioritisation - do we add more value to our organizations by curating piecemeal DevOps solutions, or by adding the next feature, unique and crucial to the success of the business?

This isn't a trade-off of process vs. pragmatism, but build vs. buy. You can spend your development dollar building bespoke DevOps solutions, but if your DevOps process doesn't afford you a unique competitive advantage over other organizations, then this isn't the best allocation of your budget. In this case, DevOps ROI is best realised by adopting a turnkey solution like Gearset, that builds on top of DX.

Finally, arguably the biggest challenge is that of piecing together tools that allow the whole team work with a single process. In our experience, the entire team needs to adopt a DevOps mindset in order to realize the benefits of this new approach. As soon as one rogue developer subverts the process and starts making changes in production, problems start to creep back in, the value of the process falls into question, and teams revert to old habits. The easiest way to avoid this is to avoid homegrown piecemeal solutions that are ever evolving to address the next gap, and instead adopt a complete DevOps solution like Gearset, designed for whole teams and built on top of the foundations laid by the DX team.

Conclusion

Release management for Salesforce has evolved into DevOps, and with the help of DX, has never been easier. Unfortunately, that's not saying much - robust release management is something the platform had never been optimized for, and DX is the start of a journey.

DX certainly goes a long way to alleviating some long-standing pains for teams of seasoned developers. Teams matching this description, however, are fairly thin on the ground, and this isn't surprising - Salesforce has long-since been the bastion of clicks-not-code and the champion of the citizen developer. This means that for the vast majority of teams, any DevOps strategy will necessarily rely on third-party tools in conjunction with these new and improved first-party DX-based building blocks for the foreseeable future.

But perhaps the most important way to look at DX right now is as a facilitating platform. The APIs the DX team are building are not only directly useful for teams to piece together their own DevOps workflows, but they lay the groundwork for third parties to build complete, accessible, solutions on top of. As time goes on, the combination of improved first-party developer tooling and tools built by the wider ecosystem will ensure that no team is left in the DevOps dark ages, and admins and developers alike will be building and releasing more effectively than ever before.

DevOps for Salesforce | **gearset**

About Gearset

This whitepaper was written by the team behind Gearset, the DevOps solution for Salesforce. The Gearset team have decades of experience in development and deployment across multiple platforms, including SQL Server, Oracle, .NET, Azure, and Salesforce. We helped build some of the leading deployment tools used by 91% of the Fortune 100 to run their release management and achieve world-class continuous delivery. We're a trusted Salesforce partner and our sole mission is to make development on Salesforce ingeniously simple. We built Gearset to solve the challenges Salesforce teams face with release management. If you're looking for a tool which can help your whole team adopt cutting-edge DevOps, whether through DX or more traditional tools, you can find more information on Gearset and start a free 30-day trial at https://gearset.com or contact us at team@gearset.com.