There are a number of common needs for triggers that are also well served by using a trigger handler. One common need is the ability to quickly disable triggers for certain users or at certain times. Trigger execution will slow data writes, so they're usually undesirable when loading large volumes of data. It's sometimes also not desirable to execute triggers when data is written as part of an integration.

Common trigger handler patterns, such as the "Comprehensive Trigger Template" by Mike Leach,[17] provide the ability to easily disable triggers for a period of time or for a particular user. Using Hierarchical Custom Settings for this purpose is useful since it allows you to determine the behavior for the entire organization or override that by profile or by user. This template also allows you to specify some processing to be delegated to `@Future` methods so they can be performed asynchronously and not count against governor limits.

The use of trigger handler patterns is probably the earliest widespread use of software design patterns in the Salesforce community.

The Domain Pattern

The use of the Domain pattern described earlier provides an alternative to using trigger handlers. The Domain pattern provides an object-oriented way to ensure that logic that should be associated with an object is always associated with that object, whether it's accessed through a trigger or through a service class.

Modular Triggers with Dependency Injection

The concept of Dependency Injection introduced earlier becomes particularly important as a way to manage trigger behavior where the logic is distributed across multiple Apex classes.

A very effective pattern is to use Custom Metadata records to track the names of classes that should be executed in a trigger context and the order in which they should execute. Multiple packages can contribute Custom Metadata records to register themselves to execute when a particular trigger runs. An admin on that org can modify the order of execution if desired.

[17] https://gist.github.com/abd3/a4a8a6b4476440adcfdea290de47948d

When a trigger runs, it first queries the Custom Metadata records (which is fast and "free" since they're stored in the platform cache), retrieves the list and order of the handler classes, and then executes them in order.

This provides loose coupling between the trigger and the handler classes, but still allows all of the execution to happen in a single transaction and to succeed or fail as a unit. John Daniel, the technical reviewer for this book, did much of the groundbreaking work to show how to modularize your codebase into packages. He created a sample project illustrating the techniques used in this process at https://github.com/ImJohnMDaniel/at4dx.

Packaging Code

The previous discussions about developing a modular, loosely coupled architecture all culminate in the topic of packaging. With the arrival of Salesforce DX, it's finally possible for enterprise developers to build packages (principally unlocked packages) on Salesforce.

Packages perform many functions and are one of the recurring themes throughout this book. Principally, they define a clear organizational unit for the codebase that makes the code easier to understand. They help ensure that your codebase is loosely coupled and thus easier to test and easier to deploy. They also help to ensure the integrity of groups of metadata, thus allowing them to be deployed consistently across orgs, including to more than one production org.

Building Salesforce packages requires that the metadata going into those packages can be decoupled from all other metadata except for that which is included in the package dependencies. This is a challenging problem, and is the reason that migration to packaging has been slow. It's also the reason I've taken the time to summarize some of the most promising techniques to achieve a modular architecture.

Building your metadata into packages doesn't perfectly guarantee that your code is loosely coupled, and you can encounter unexpected behavior if the versions of your packages are out of sync between different orgs. Nevertheless, keeping track of package version numbers is infinitely easier than keeping track of subtle variations between unpackaged metadata spread across many orgs.

Salesforce DX allows you to specify package dependencies in the `sfdx-project.json` file as shown in Listing 5-5. Note there are two valid syntaxes. The first uses an alias like `industry@0.1.0.12546` to point to an ID; this syntax is used when the package is not included in the same project folder. The second syntax is to specify both the `package` and a `versionNumber`; this syntax is only valid for packages that exist on the same Dev Hub.

Listing 5-5. An `sfdx-project.json` file showing package dependencies

```
"packageDirectories": [
  {
    "path": "force-app/healthcase",
    "package": "healthcare",
    "versionName": "ver 0.1",
    "versionNumber": "0.1.0.NEXT",
    "dependencies": [
      {
        "package": "industry@0.1.0.12546"
      },
      {
        "package": "schema",
        "versionNumber": "0.1.0.LATEST"
      }
    ]
  },
  {
    "path": "force-app/schema",
    "package": "schema",
    "versionName": "ver 0.1",
    "versionNumber": "0.1.0.NEXT",
    "default": true
  }
],
"packageAliases": {
  "industry@0.1.0.12546": "04t1T000000703YQAQ",
  "schema": "0Ho6F000000XZIpSAO"
}
```

Specifying package dependencies is one of the most important aspects of packaging. Figure 5-6 shows the actual metadata dependencies in one Salesforce org I worked on (this org was large, but by no means the most complex I've seen). Each node in the graph represents a piece of metadata such as a class or field, and each edge in the graph represents a dependency (small arrows show the direction of the dependency).

This level of complexity is impossible for humans to reason about, which means that it's difficult or impossible to understand the implications of changing a piece of metadata. When making changes is risky, innovation is stifled and teams delay activities like refactoring or reorganizing the code.

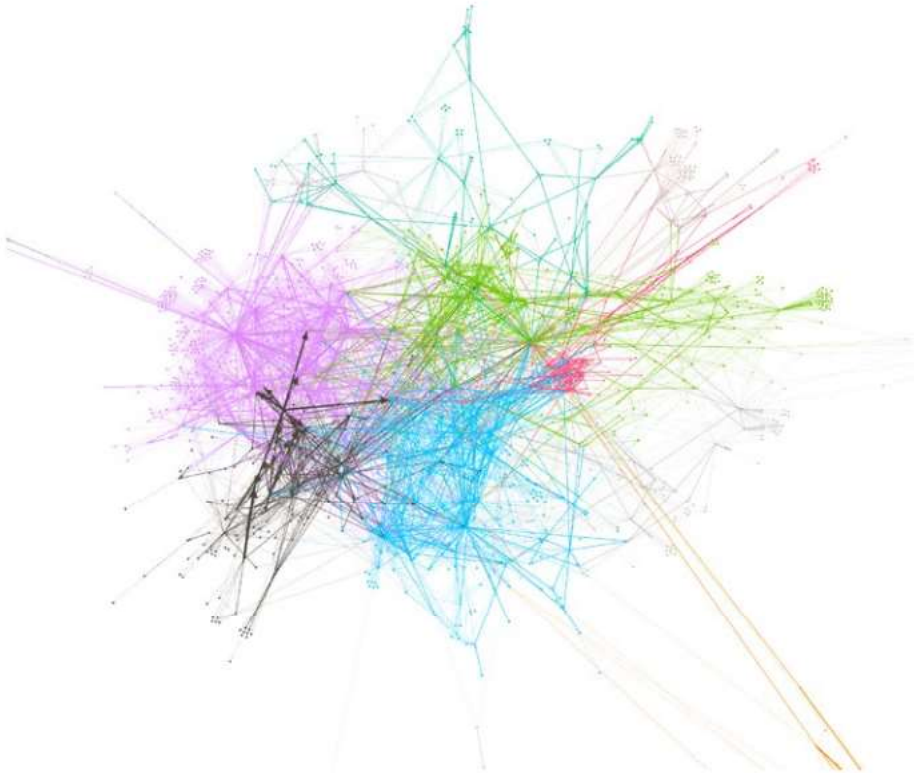

Figure 5-6. *Actual metadata dependencies in a large Salesforce org*

Contrast this with Figure 5-7, which shows the same org's metadata after refactoring it into packages and the interrelationships between packages. Each of these packages may contain their own complexity, but that complexity is hidden from view and somewhat irrelevant to other packages. This means that the implications of making a change to one of the packages are much clearer.

Since in Figure 5-7 the "base-metadata" package is depended on (directly or indirectly) by all of the other packages, changes to it should be made with care. But since no other packages depend on the "Account Planning" or "Customer Community" packages, the teams responsible for those packages don't need to worry about causing side effects in other packages. There is still complexity in this diagram, but it's trivial compared to

the metadata dependencies in unpackaged metadata shown in Figure 5-6. Simplifying dependencies in this way makes it much easier to understand the risks associated with changing packages.

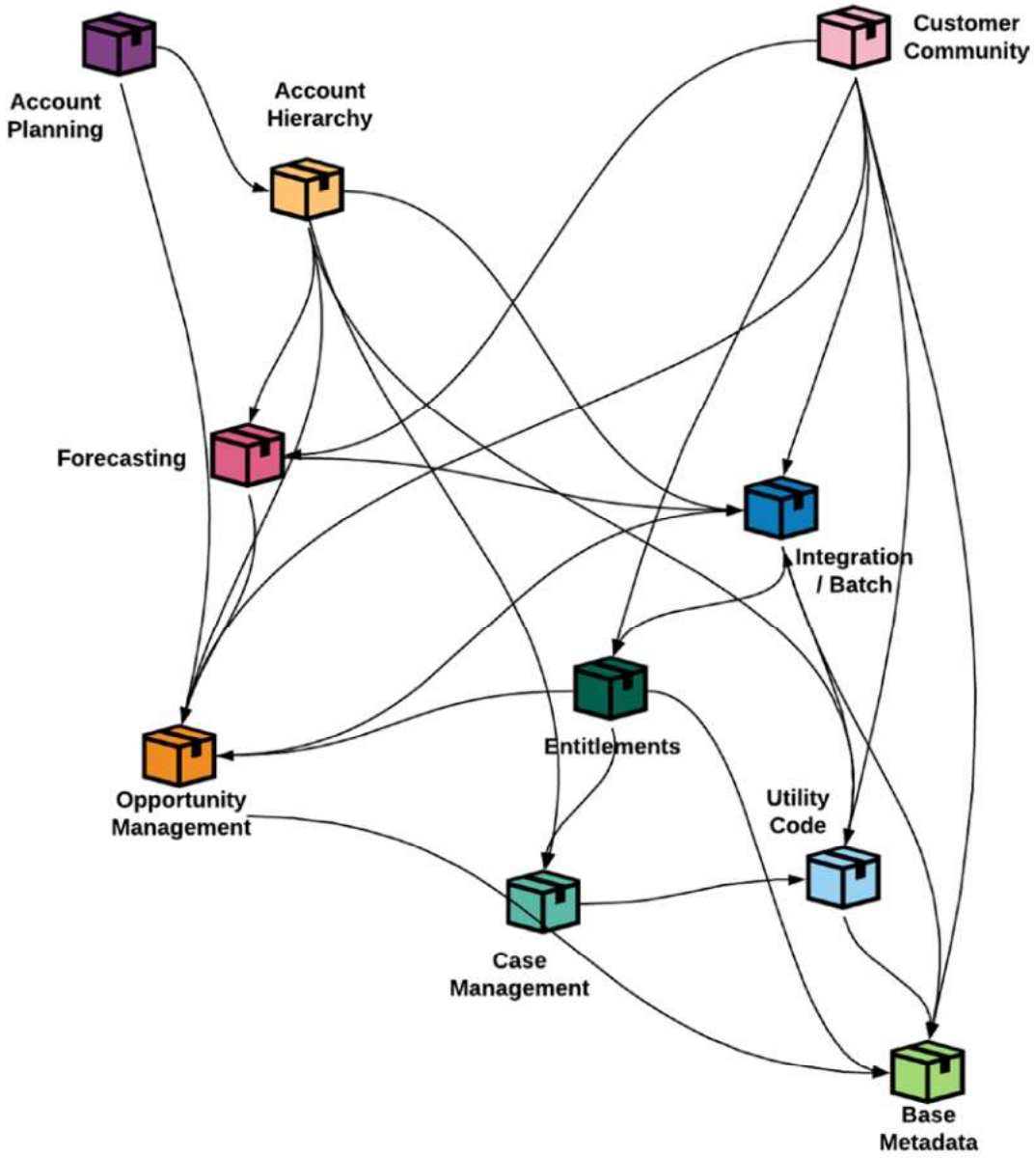

Figure 5-7. *A depiction of package dependencies*

Summary

Refactoring metadata so that it can be built into unlocked packages is the single most challenging aspect of adopting Salesforce DX. But it is also the most beneficial.

To some degree, the refactoring process can be undertaken gradually. If you are beginning work on a new application with limited dependencies on other metadata, that might provide an opportunity to begin working with unlocked packages. There is some tooling required to make this process easier, much of which is described in later chapters. There are many human factors involved in such an effort as well.

If your team works on a relatively independent part of the Salesforce codebase, it may be possible to get started independently. But in most cases, you'll soon find that you need to store and modify some common metadata like Account fields that are also shared with other teams. Therefore, typically the entire development organization first needs to move together onto a common delivery pipeline to deploy metadata at the org level.

The process of setting up a delivery pipeline, working with version control, setting up CI/CD, automated testing, deployments, and releases is the topic of Part 3: Innovation Delivery. Even once your entire org is refactored into packages, you'll still need to manage some metadata at the org level; therefore, it's beneficial to set up an org-level CI/CD pipeline from the very beginning. Ideally you should set this up using the Salesforce DX file format so you can benefit from easier merges, better filenames, and the ability to create folders for metadata.

You can then set up one or more delivery pipelines for packages and begin the process of migrating metadata out of your org-level repository and into your package repository(s). Keep returning to and experimenting with the architectural techniques introduced in this chapter as you go through that process. It is not easy to refactor a codebase in this way, and so it's also important that you keep clear goals for the process for both the development and business teams.

Your long-term goal in this process is to continually increase your team's ability to develop and release functionality for the business quickly, safely, and sustainably. Keep metrics on your starting-off point, and periodically check in on whether your refactoring efforts are enabling you to deploy more quickly and reduce the incidence of failures. DevOps is a process of continual improvement. Your architecture is a key factor in enabling or inhibiting your team's ability to continually improve.

PART III

Innovation Delivery

Software delivery is the phase of the software development lifecycle in between development and operation. It's the phase in between something being created and that thing being used. In this book, I refer to this phase as "Innovation Delivery" to emphasize that much of what we're delivering in Salesforce are non-code changes. To use the analogy of groceries, things are first grown or manufactured, then they're picked, shipped, and stored, and finally they are purchased and consumed. Software delivery is analogous to the picking, shipping, and storing phase.

This analogy is important in illustrating that this middle phase doesn't add much value, and in fact it often adds risk and waste. The longer things are held in storage, and the longer it takes to ship them, the shorter their eventual useful life and the more chance that they'll go bad or get out of date. Whereas manufacturing adds value (transforms raw materials into a valuable finished product), shipping rarely adds much value. It simply makes that valuable thing available for use.

The essence of Lean management is to identify and eliminate waste from the system. To eliminate waste, you first need to understand the system as a value chain, a series of interconnected steps, each of which contributes value to the finished product. Within a value chain, there are typically steps that do not add any value. Those steps are deemed "waste." Since DevOps incorporates Lean management, we can say that the goal of DevOps is also to eliminate waste in a system. And much of this waste can be found in the Innovation Delivery phase, which is why DevOps puts such strong emphasis on understanding and improving this phase.

Innovation delivery consists of environment management, creating a delivery pipeline, testing, deploying, and releasing. Although we've spent considerable time in this book explaining development techniques that enable DevOps, we've done so largely in the spirit of enabling success in this delivery phase. It is in this phase that DevOps techniques such as continuous delivery really shine, and it's this phase that is the heart of this book and the phase where the innovations of Salesforce DX have the biggest impact.

CHAPTER 6

Environment Management

Careful application development demands different environments to play different roles such as development, testing, training, and production. The software delivery lifecycle is largely a process of promoting changes through these environments, and thus it's important to begin with an overview of the different types of Salesforce environments and how to manage them.

An Org Is Not a Server

Although this is obvious to Salesforce veterans, it's an important clarification to make for anyone new to the platform. A Salesforce org is not a server. It's also not something that you can ever run on your local machine or on your corporate network. It's also not something that you can provision for yourself on a cloud infrastructure provider such as Google Cloud or AWS.

All Salesforce development, testing, and use take place in a Salesforce org, and Salesforce orgs can only ever be provisioned by Salesforce, in one of their own data centers, accessible only via the public Internet. Got it?

We'll discuss more of the underlying architecture in "A Behind-the-scenes Look at Salesforce Orgs." The way that orgs exist was a strategic decision by Salesforce that ensures that they can control the infrastructure, the upgrade process, and access to the underlying intellectual property. There are never outdated versions of Salesforce running behind a corporate firewall. Throughout most of the year, there is only one version of Salesforce running in the world. During the Seasonal Upgrade Period

© Andrew Davis 2019
A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_6

(described later in the section on "The Salesforce Upgrade Cycle") and during times when patch releases are being rolled out, there are two versions of Salesforce running, as orgs are upgraded in batches.

Another reason why orgs exist in this way is that there are no orgs. Sound like a Buddhist paradox? What I mean by this is that not only is an org not a server; it's not a virtual machine; it's not a cluster; and it's not a container. It's actually just a unique ID that allows Salesforce to isolate the data and metadata that belong to one customer from the data and metadata of all other customers. In general, Salesforce can be seen as a single massive application, running on a single massive database, with all customers' data in that same database, segregated by a single Org ID such as `00DB00000001Tax`.

Many layers of carefully crafted security, application, and database optimization are then used to generate a unique experience in each Salesforce org, so that it appears to function as a completely independent instance. Even Salesforce support have no access to data within a customer's org unless they are explicitly granted access on behalf of a specific user for a period of time.

Thus, when we talk about environment management, the process is somewhat different from the process on other platforms. It's nevertheless still important to distinguish different types of org and to provision different orgs for different purposes.

Different Types of Orgs

For practical purposes, there are only three types of Salesforce orgs that most people need to know about: production orgs, sandboxes, and scratch orgs. These different types of orgs behave in almost identical ways, but have different limits (such as the amount of data and records they can hold) and may have different features enabled.

Salesforce production orgs are the ones you pay for. They come with different features depending on which edition and licenses you buy. The first distinction is between the different editions of Salesforce such as Essentials, Professional, Enterprise, and Unlimited. Each edition is progressively more expensive but offers increasing capabilities and higher limits on data storage, number of sandboxes, and so on.

There are also free editions used for training and demonstration purposes such as Developer Edition and Partner Edition. A single org is always of a particular edition. Since Essentials and Professional editions do not support custom code or API access, they are rarely used by large organizations. Enterprise edition is the most common.

Orgs are priced on a per-user basis, and there are also different user license types available, the most common being Sales Cloud User, Service Cloud User, and Platform User licenses. Each user has exactly one user license. Whereas the edition unlocks certain capabilities (such as API access) across the entire org, user licenses allow different users to access different types of data and capabilities. A common scenario would be for a company to have an Enterprise Edition org in which the salespeople are assigned Sales Cloud licenses, and the customer support representatives and their managers are assigned Service Cloud licenses. In addition, there may be employees who are assigned only Platform licenses so they can make use of certain custom Salesforce applications. Meanwhile all employees of the company might have access to Chatter Free licenses so that they could at least collaborate using Chatter, and customers of the company may have licenses to use a self-service community created from that Salesforce org.

Figure 6-1 shows the 2019 pricing for Sales Cloud user licenses.

Sales Cloud

WATCH DEMO

Overview

Features

Related Products

Pricing

By Role

Customer Stories

Resources

FAQ

QUESTIONS?

1-800-667-6389

CONTACT US

Sales Cloud Pricing

Sell faster and smarter with any of our fully customizable CRM editions.

Salesforce Essentials	Lightning Professional	Lightning Enterprise	Lightning Unlimited
		MOST POPULAR	
Out-of-the-box CRM	Complete CRM for any size team	Deeply customizable sales CRM for your business	Unlimited CRM power and support
$ 25 USD/user/month* (billed annually)	$ 75 USD/user/month* (billed annually)	$ 150 USD/user/month* (billed annually)	$ 300 USD/user/month* (billed annually)
TRY FOR FREE	TRY FOR FREE	TRY FOR FREE	TRY FOR FREE

+ EXPAND ALL ● ALL FEATURES ○ SOME FEATURES — NOT INCLUDED $ ADDITIONAL COST

TRY FOR FREE	Salesforce Essentials $ 25 USD/user/month	Lightning Professional $ 75 USD/user/month	Lightning Enterprise $ 150 USD/user/month	Lightning Unlimited $ 300 USD/user/month
Find and manage leads better.	○	○	●	●
Lead Management	●	●	●	●
Lead Assignment and Routing	●	●	●	●
Rules-based lead scoring		●	●	●
AI-Powered Lead Scoring			$	$
Duplicate Blocking	●	●	●	●
Web-to-Lead Capture	●	●	●	●

Figure 6-1. *The Sales Cloud user pricing page*

Developer Edition orgs are a special type of production org that are free and intended for use by developers. You can sign up for a Developer Edition org at `https://developer.salesforce.com/signup`. Developer Edition orgs have access to almost every capability that is available on Salesforce, making them ideal places for developers and admins to learn and experiment. Available features include Sales Cloud, Service Cloud, and Community Cloud as well as add-ons such as Work.com, Identity, Knowledge, Data.com, and Live Agent that would normally be paid add-ons to a Salesforce org. Developer Edition orgs have strict limits on the amount of data stored, API requests, and active users, so they can't be used for any significant business activity. Developer Edition orgs are updated on the same schedule as Salesforce production orgs. Salesforce partners can also get access to Partner Edition orgs, which are identical to Developer Edition orgs but with higher limits.

The second main type of Salesforce orgs are sandboxes. Sandboxes are long-lived development, testing, or training environments that are associated with a particular production org. Until recently, sandboxes were necessarily created as clones of the production org (having the same metadata and configuration as the production org). It is now also possible to clone one sandbox from another, which opens up entirely new workflows (see the section "Cloning and Refreshing Sandboxes").

Sandboxes also come in different sizes, each with varying levels of data storage: Developer, Developer Pro, Partial Copy, and Full Copy. The most inexpensive type of sandbox is a Developer sandbox (not to be confused with Developer Edition orgs, which are actually production orgs). Developer sandboxes cloned from the production org have the exact same metadata as the production org but do not contain *any* data from that org. Developer sandboxes that are cloned from other Developer sandboxes *do* contain a copy of the data from the source org, which allows an organization to establish certain orgs as templates from which other orgs can be cloned.

Developer Pro sandboxes have slightly higher data storage limits. Partial Copy sandboxes have substantially higher data storage limits than Developer sandboxes and also can receive a partial subset of production data copied into them when they are refreshed from production. Full Copy sandboxes allow the entire database from a production org to be replicated and are typically used as staging orgs where functionality can be tested against real data conditions prior to being deployed to production. Different editions of Salesforce come with different numbers and types of sandboxes, but customers can pay to get more. For example, Unlimited Edition includes one free full sandbox, for other editions it's a paid add-on.

The final type of org is also the newest type: scratch orgs. Scratch orgs are a key part of the Salesforce DX workflow. Scratch orgs are similar to sandboxes, since they are necessarily created from and associated with a production org. A production org that has been enabled to create scratch orgs is called a Dev Hub and is discussed more in the next section. Like sandboxes, scratch orgs are used for development, testing, training, or demos but, unlike sandboxes, scratch orgs contain no data or metadata from the production org and are short-lived orgs that are deleted after a period of time (7 days by default).

Scratch orgs also possess two important characteristics that make them ideal for a DevOps workflow: their characteristics (Edition, Features, Settings, Name) can be defined in a configuration file, and their metadata can be easily synced with version control. Even their short lifespans should be viewed as a feature, not a limitation: this design constraint forces teams to not rely on an org being their source of truth, but instead to capture all relevant configuration in version control. We'll discuss scratch orgs in more detail later in this chapter, but it's important to note that they do not replace sandboxes but rather complement them. There are disadvantages to using sandboxes as development environments, but they are still useful as long-lived testing and training environments.

Salesforce recently announced a new type of sandbox, Lightning Developer Pro sandboxes. These have identical characteristics to Dev Pro sandboxes, but can be created and destroyed using API commands, similar to scratch orgs. Unlike scratch orgs, these orgs begin as a clone of an existing org (generally your production org), which means they have all of the features, settings, managed packages, and metadata of their source org. Thus they don't require teams to refactor their orgs into packages, but nevertheless provide a short-lived development environment that can be used to make and test incremental changes.

Which Org Should You Develop In?

Given these different types of orgs, how and where should you develop new functionality?

Why Not Develop in Production?

Small organizations, organizations that are new to Salesforce, and those with loose governance and less restrictive security requirements might be doing development work directly in production. Most types of Salesforce "development" don't involve code, and

changes such as adding a new field to an object are almost always safe. Even code-based development in Salesforce is not as risky as directly changing code on, say, a production web server. For example, in most cases a coding error in one Apex class will only impact a limited part of your org, whereas in a traditional application, a coding error could cause the entire app or site to fail.

It's partly for this reason that most organizations persist in doing some "development" directly in production. However, there are always dangers to doing this.

It's entirely possible to cook your meals and eat them right out of the pot, standing in the kitchen; it's efficient and minimizes the time you spend washing dishes. Most of the time though, people opt to move from the kitchen to the dining table to enjoy what's been prepared, even if they live alone. The more important the situation and the more people are involved, the more important it is to segregate these activities. Even the cleanest kitchens get messy quickly. Thus in most restaurants, chefs prepare food in the kitchen, waiters deliver it to the table, and patrons enjoy it without ever seeing where it was made.

Just as you can eat straight out of the pot, you can develop directly in production. But development environments are no more clean than kitchens, developer changes are no more safe than working with knives or fire, and an org that changes or breaks unpredictably is no more pleasant for your users than the noise and bustle of a short-order grill.

Just as a well-run restaurant develops a smooth flow of sending orders to the kitchen and food to the table, it's important to optimize the process of building in a development environment and deploying to production.

What are the dangers of developing directly in production?

1. Making changes without testing can lead to application failures, data loss, or corruption.
2. The risk of making changes inhibits innovations that might bring benefit but which require iteration and testing.
3. Changes are hard to track. Most metadata shows who last modified it and when, but does not show what changed or the history of changes. Salesforce provides a setup audit trail, but it does not show much detail.
4. Untracked changes are hard to reverse. It's difficult or impossible to reverse changes when you don't have their history.

5. Changes in production are not propagated to other testing or development orgs unless those orgs are refreshed, which wipes out any changes in them.
6. This practice may violate compliance laws such as SOX, HIPAA, or PCI-DSS. Instead, changes should be made in a development environment, tracked in version control, and deployed to production using a CI/CD system.

Developing in Sandboxes

As mentioned in Chapter 2: Salesforce, prior to the arrival of Salesforce DX, the normal workflow for Salesforce was for a team to do all of their development in one or more Developer sandbox and then to migrate those changes to the testing and production org(s). As of this writing, that is still the most common workflow for teams, since for complex orgs the migration to Salesforce DX usually requires some care and time.

When developing using sandboxes, the team creates one or more Developer or Developer Pro sandboxes, each of which contains an exact copy of production metadata. Teams then make their code or config changes in that org and use some mechanism to capture those changes and deploy them to testing, training, and production orgs.

Salesforce's Development Lifecycle Guide (now deprecated) recommended creating one Developer sandbox for each developer, and some guides still recommend this. The benefit of having one Developer sandbox per developer is that the sandbox contains only the changes made by that developer. It therefore gives the developer total control over that org and makes it relatively easy to identify all of the metadata touched during development, so that metadata can be moved to other orgs.

There are, however, major disadvantages to every developer having their own sandbox. Synchronizing changes between orgs requires deployments, and performing multidirectional deployments between many developer orgs requires substantial time, repetitive manual changes, and deployment error resolution and brings ample risk of overwrites. This generally prevents continuous integration, since it effectively isolates each developer in orgs that invariably become more different from one another over time.

By contrast, forcing developers to develop in a single, shared Developer sandbox ensures that each developer's work is continuously being integrated with the work of every other developer. Unfortunately, it also makes it difficult to isolate which changes belong to which feature, and so difficult to deploy features independently.

Developing in sandboxes is Org-based development, wherein a Salesforce org itself is the source of truth. If all development is done in a single, shared Developer sandbox, that org becomes the unquestionable source for the latest state of the config and code. If developers make conflicting changes, those conflicts are generally recognized and resolved very quickly. This also brings a risk of having your work overwritten, which gives plenty of incentive to start using version control!

Integrating conflicting changes requires developers to pause and communicate to resolve the conflict. When developers communicate and resolve conflicts on a daily basis, during the actual development process, each developer can more easily remember and articulate the reasons why their changes are important and so can more easily find a resolution. Developing in independent sandboxes over weeks or months causes potential conflicts to accumulate, while developers gradually forget the detailed reasons behind each change. This makes integration painful and far more risky. It also makes it very difficult for anyone to make small iterative changes to improve the entire codebase (refactoring), since old versions of the code will creep back in during integration.

Isolating teams into separate sandboxes brings the same risks as isolating individual developers in their own sandboxes. Parallel work streams, such as simultaneous Sales Cloud and Service Cloud implementations, will face the same problems if they delay integrating their codebases. It's tempting to have teams develop in parallel so they avoid the risk of conflicting changes. But delaying conflict resolution until close to go-live is a far riskier strategy.

Developing in a shared developer sandbox thus ensures a type of continuous integration, a recognized best practice in software development. There are nevertheless major disadvantages to developing in sandboxes compared to using scratch orgs for this purpose.

The Disadvantages of Developing in Sandboxes

The purpose of developing is to deliver innovations that benefit production users. The purpose of using a separate development environment is to innovate safely. Safe innovation also requires that each change be tracked so that it can always be rolled back. And in general, small, frequent innovations are safer to deploy and easier to debug. Developing in this way implies the use of version control and continuous delivery.

It is possible to use version control and continuous delivery when doing sandbox-based development, by capturing changes from the development environments in version control and then using continuous delivery to update testing

and production orgs. But in this scenario, version control necessarily lags behind the development org, so development sandboxes are the actual source of truth.

Sandboxes are basically immortal. Unless you refresh or delete them, they will remain for as long as your production org remains. They always start as a replica of their source org (usually production), and they continue to accumulate changes until they are refreshed or deleted. But because the sandbox itself holds the latest features under development, refreshing the org means losing any changes that you have not already saved or deployed. Development has to stop (or move to a new sandbox), and you have to be able to copy or restore all of the work that was under development. Unless teams are particularly disciplined about refreshing developer sandboxes, the simplest, safest option chosen by most teams is to never refresh the development org. But with every passing day, the Developer sandbox becomes further and further out of sync with production.

Development is a messy process. Just as a craftsman's wood shop accumulates sawdust and scraps, development environments accumulate cruft—duplicate fields, half-finished customizations that were eventually abandoned, and so on. Large organizations have enormous amounts of customizations and complex architectures. Long-lived Developer sandboxes contain all that complexity, plus the accretion of months or years of half-baked or unneeded changes.

Even if a team is using version control to deploy changes to their test and production orgs, many changes in the development org may never be tracked in version control. There are also some changes that can be made in an org that may be critical for functionality but that can't be tracked in version control. This can lead to difficult to debug problems when you finally attempt to deploy a feature or application that has been under development for a long time.

The Benefits of Developing in Scratch Orgs

Scratch orgs offer enormous benefits for team development. But the team needs to understand these benefits and the implied changes to their workflow if they are to become comfortable developing in scratch orgs.

For those familiar with developing in long-lived sandboxes, developing in a scratch org, which expires after several days, is disconcerting. Developing in such an ephemeral environment is truly a paradigm shift, but you should understand this as a feature, not a bug. The short lifespan of a scratch org is a design constraint that forces developers to

not rely on the org, but instead to persist their work in version control. There is no better motivator for developers to use version control than the threat that their development environment will soon disappear.

Scratch orgs can be configured to persist for up to 30 days, but as a rule teams should strive to create automated setup processes that make creating new orgs a fast and trivial process.

Unlike sandboxes, scratch orgs do not inherit any data, metadata, features, or preferences from the Dev Hub that creates them. As explained later in this chapter, Salesforce is planning to release capabilities for creating scratch orgs from Snapshots and creating scratch orgs from an Org Shape, but in general scratch orgs are created entirely from configuration stored in version control. That means there's no ambiguity about what features, settings, packages, or metadata are required to create an application in a scratch org. Because everything is recorded in version control, customizations can be easily recreated in another scratch org or deployed to other orgs as metadata or as packaged applications.

Scratch orgs also support "source synchronization," which means you can just run `sfdx force:source:pull` and `sfdx force:source:push` to synchronize metadata between version control and your org. With org-based development, developers use metadata retrieve commands to "pull" metadata from these orgs and metadata deploy commands to "push" metadata to these orgs. But deploying your team's changes into a private Developer sandbox requires struggling through deployment errors and risks overwriting your own customizations. And picking your own changes out of the fast-changing ocean of unpackaged metadata in a shared Developer sandbox requires deep expertise in the different types of Salesforce metadata.

Source synchronization means that you do not need to be an expert in Salesforce metadata. Open a scratch org, make changes, pull down the source for those changes, review it to remove any cruft or unneeded changes, commit it, and then merge your work with the team's code repository. Every developer can have a private development environment, can track their changes with confidence, and easily share customizations with the rest of the team.

Since the entire team is generally pushing and pulling source many times a day, deployment errors and conflicts are spotted very quickly. Without this workflow, it can take days or weeks before deployment errors reveal to teams that their commits to version control contained missing or invalid metadata.

Scratch orgs complement sandboxes but do not replace them. Because sandboxes are production-like environments, they remain useful as long-running test environments. Sandboxes for integration testing can be configured with stable integrations to external systems. Sandboxes for testing production hotfixes can be refreshed as frequently as needed. Sandboxes for training can be recreated with real production data in a Partial Copy sandbox. And Full Copy sandboxes can be used as staging environments to ensure that customizations are performant and able to handle the diversity of production data.

As of this writing, scratch org snapshots are not yet available. This means that it can be challenging or impossible to quickly provision scratch orgs for teams building on top of large managed packages, extensive and fast-changing configuration data, or settings that are not supported in scratch orgs. My view and hope is that this is a transitional time and that developing in scratch orgs will gradually become viable for even the most complex orgs.

Meeting in the Middle—A Sandbox Cloning Workflow

Scratch orgs are built "from the ground up," beginning with an empty org. Scratch orgs give you complete confidence that you know what's in your org, why it's there, and that you can recreate it whenever necessary. But organizations may struggle to build packages and record all the org preferences from their existing orgs. And complex orgs can require more than an hour to install the necessary packages, additional time to deploy metadata, and significant data loads to establish meaningful sample data.

Sandboxes are established "from the top down," in that they're always cloned from an existing org, complete with preferences, packages, and metadata. That means teams can immediately get started using them for development, testing, or training.

Salesforce DX is a growing technology, and the Salesforce DX team is working to enable a wider array of options to enable efficient workflows for development and release. Enabling scratch org snapshots allows teams to build a template org from the ground up and then create working orgs from that template very quickly. Similarly, enabling fast sandbox cloning, source synchronization in sandboxes, and automated authentication means that sandboxes can also begin to be part of an efficient and automated workflow.

For teams working on complex orgs, a sandbox cloning workflow can provide a mechanism for each developer to have their own environment that can be regularly integrated with changes from the rest of the team.

The key to this is to establish an integration sandbox that receives deployments from all of the developers and to regularly recreate development environments by cloning this integration sandbox. The cloning and authentication of this integration environment can be automated using the Salesforce CLI as explained in "Postcreation Sandbox Login." Changes that developers make in their individual sandboxes should be tracked in version control and deployed to the integration environment on an ongoing basis. Teams can use trunk-based development to ensure that their changes are expedited to the integration org, or feature branches to ensure that everyone's changes can be validated and reviewed. But the key to efficiency with this workflow is to regularly recreate those developer sandboxes to ensure they stay up to date and don't accumulate cruft.

Org Access and Security

Salesforce Orgs are accessible (and secured) in many ways. The most common methods of access are through the login user interface (`https://login.salesforce.com`, or `https://test.salesforce.com` for sandboxes and scratch orgs) and via APIs. Whether users log in interactively or via the API, access is always associated with a particular username and limited by that user's permissions. Usernames must be unique across all Salesforce orgs and are in the form of email addresses, although they do not need to correlate with actual email addresses. Users can be created and deactivated, but can never be deleted. Whenever a sandbox is created or refreshed, user accounts are created in that sandbox for every user that exists in the source org from which it was created. Sandbox usernames are autogenerated by appending the sandbox name to the source username. For example, if a user's production username was user1@production.org, when a sandbox called UAT is created from that production org, it will contain a user account with username user1@production.org.UAT. User's email addresses are automatically modified to prevent sandboxes from accidentally emailing users (and possibly confusing users). Email sending is also disabled by default in sandboxes.

Sandbox user accounts have the same profile, licenses, permission sets, and password as the user account in the source org. This means that if a user knows the name of a sandbox, they can log in to that sandbox by going to `https://test.salesforce.com`, appending the sandbox name to their production username, and using their production password. Users who log in using SSO may not even know their production passwords, however, and SSO has to be set up separately for each sandbox. The correlation between sandbox user accounts and production user accounts provides

a convenient way for admins to direct users to log in to a sandbox for testing or training if needed. Because a user's permissions are also copied from the source org, even on full copy sandboxes the user will not be able to access data that they were not allowed access to in production. But it is important to understand this behavior, and data security in full and partial copy sandboxes should be considered alongside data security in production orgs.

Scratch orgs behave in a very different way. Scratch orgs are not created from a production org or sandbox. Because they contain no metadata or data from a long-lived org, their security is not as big a concern. The user who creates a scratch org automatically gets sysadmin access in that org, which allows them to use that environment for development without any restrictions. A typical first step after creating a scratch org is to populate that scratch org with metadata and test data stored in version control. But access to view and edit that metadata and test data is controlled by the version control system itself; Salesforce doesn't create any additional restrictions.

Just as scratch orgs don't contain any metadata or data from a source org, they also don't contain any user accounts aside from a single sysadmin user. It's possible however to use Salesforce CLI commands to create user accounts in a scratch org so that applications can be reviewed and tested from the point of view of users with different permissions.

The Dev Hub

Salesforce DX capabilities such as creating scratch orgs and publishing versions of Unlocked or Managed packages depend on a Dev Hub. A Dev Hub is a Salesforce production org that has special capabilities enabled. All paid Salesforce orgs and free Developer Edition Orgs can function as Dev Hubs. For training purposes, you can use your own Developer Edition org, but this allows only a small number of scratch orgs and packages. For business work, you should designate a production org that the development team has access to and enable Dev Hub on that org. For example, Appirio's internal development teams use Appirio's production org as their Dev Hub, but for customer projects we used the customer's production org as the Dev Hub. Developers should never be given access to edit metadata in production orgs (see "Locking Everybody Out" in Chapter 12: Making It Better), but there is no security risk with giving them permission to use the production org as a Dev Hub.

Permissions Needed to Use the Dev Hub

To use an org as a Dev Hub, that capability first needs to be enabled by going to `Setup UI ➤ Dev Hub`. For a user to make use of the Dev Hub, they then need several specific permissions. Administrators have these permissions automatically, but you shouldn't give your developers Admin rights in production. Add the necessary permissions to a permission set or profile, following the steps in the Salesforce DX Setup Guide. Permissions include

- Create and delete scratch orgs
- Create and update second-generation packages

When a package version is created, it is in the "beta" status and cannot be installed in a production org. To enable users to mark a package version as "Released" so that it can be installed in a production org, you need to enable the system permission `Promote a package version to released`.

What if developers don't have access to the production orgs? In these cases, Salesforce support can enable "Free Limited Access Licenses" in that production org. These secure licenses allow access to the Dev Hub, but do not allow any other data, login, or API access. Developers can be safely assigned this type of user license, even when they have no other access.

API-Based Authentication

While most user interactions with Salesforce involve users logging in manually and then viewing and editing the org interactively, automated processes are usually based on API access. Salesforce provides a very rich set of REST and SOAP-based APIs to access data and metadata and to trigger operations. Their robust API has allowed for the creation of a vast number of integrations from other enterprise software providers, as well as from tools specific to Salesforce. Since the technical processes in DevOps focus largely on automation, you'll be directly or indirectly exploiting these APIs as you refine your development workflow.

The most common API workflow involves using the REST API based on OAuth 2.0 authentication. The OAuth 2.0 flow for authenticating against Salesforce can be briefly summarized as follows:

1. Prompted by user actions, a local tool or third-party service requests access to Salesforce on the user's behalf.
2. The user is redirected to Salesforce and logs in.
3. Salesforce then presents an *authorization* screen similar to Figure 6-2 asking whether the third-party tool should be allowed access to Salesforce on the user's behalf. This screen specifies which permissions ("grants") are being requested on behalf of the user. Some apps require limited permissions such as "Access your basic information" (e.g., name, username, email), whereas others may request more permissions. Apply appropriate care when authorizing an app—you may be giving it permission to access all of your data and manage Salesforce on your behalf.
4. If you allow access, Salesforce will then send the third-party application an **Access Token** on your behalf. That access token is used to read data and perform actions on Salesforce on your behalf, subject to the limits of whatever permissions you gave in the authorization page.
5. Importantly, if you approve the permission "Perform requests on your behalf at any time," the application will be given a **Refresh Token** that allows it to reconnect to Salesforce at any future time. The Refresh Token is used to request a fresh Access Token whenever an Access Token has expired.

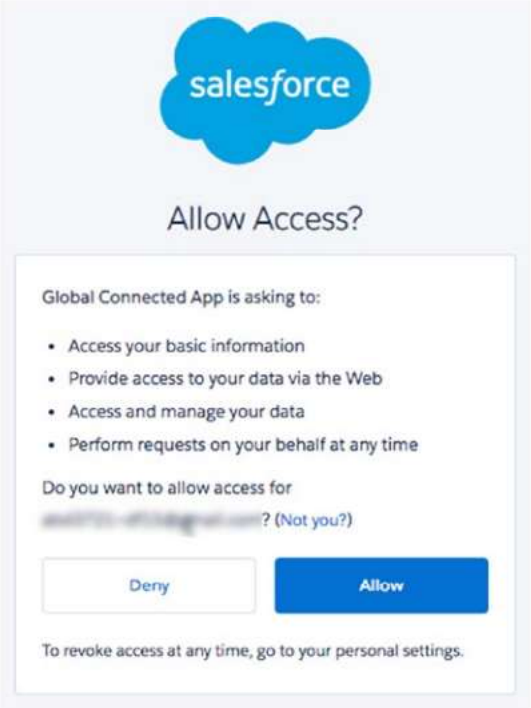

Figure 6-2. *An OAuth authorization prompt requesting full access to your account*

At any point, you can view the applications that were authorized using your credentials by visiting `Settings` ➤ `My Personal Information` ➤ `Connections`. To see the OAuth connections for all users across the org, admins can go to `Setup UI` ➤ `Apps` ➤ `Connected Apps` ➤ `Connected Apps OAuth Usage`.

OAuth 2.0 authentication is considered more secure than username/password authentication, because the permissions for an OAuth Connected App can be tuned carefully, monitored, and revoked if needed. Sharing your Salesforce username and password with a third-party app is fundamentally less secure, so when given the chance you should give strong preference to tools that use OAuth.

For long-running integrations between Salesforce and another system, you should always create and use an integration user and perform the authentications as the integration user. Using an integration user ensures that the connection is not disabled as a result of the user who initially set up the integration leaving the organization. It also enables multiple users to monitor the activity of the integration user account.

Salesforce DX Org Authorizations

In the context of Salesforce DX, API authentication is generally handled using the Salesforce CLI. Using the Salesforce CLI, you can authenticate to an org, and save your org connection with a friendly alias such as "UAT." You can then perform commands using those aliases such as `sfdx force:org:open -u UAT`.

The Salesforce CLI provides developers with many different ways of authorizing Salesforce orgs. Some of these methods are more appropriate for manual use by developers, whereas others are designed to allow CI systems to autonomously perform actions on Salesforce orgs.

The main **auth** commands the Salesforce CLI provides are as follows.

force:auth:web:login

This is the typical method for authorizing orgs when working with Salesforce CLI commands locally. It opens a browser tab to allow you to input your credentials to authorize the org. This type of login is generally never used directly in automated processes such as continuous delivery. But by first authorizing an org locally, you can then access an Auth URL for that org that can be used in automated processes.

force:auth:sfdxurl:store

The simplest way to enable API access to Salesforce as part of an automated workflow is to take advantage of a special string called an Auth URL that the Salesforce CLI generates.

This one URL contains all the information necessary to access an org on a user's behalf, including the endpoint URL, OAuth Client ID, and refresh token.

Auth URLs have the format: `force://clientId:clientSecret:refreshToken@instanceUrl`

To get the Auth URL for an org, follow these steps:

1. Authorize the org whose Auth URL you need using `sfdx force:auth:web:login -a OrgAlias`. By default, this command opens https://login.salesforce.com. If your org uses a custom domain or you need to authorize a sandbox, specify the `-r` parameter with the custom domain.

```
$ sfdx force:auth:web:login -a OrgAlias -r https://test.
salesforce.com
$ sfdx force:auth:web:login -a OrgAlias -r https://myDomain.
my.salesforce.com
```

2. Log in with your credentials and authorize the Global Connected App.
3. Once you've successfully authorized the org, close that browser tab and run `sfdx force:org:display -u <OrgAlias> --verbose` command (substituting the appropriate OrgAlias) as shown in Figure 6-3.
4. This will display the Auth URL for the org.
5. Your CI system will have some mechanism for storing secrets. You can then add that Auth URL as a secret variable in your CI system to connect to that org. It is extremely important to secure this Auth URL, and any other access credentials, as you would a password. CI systems should all provide a mechanism for storing secrets. These should never be hardcoded or included in the repository.
6. In a CI job, you will first need to write that secret variable to a temporary file, then read it into the Salesforce CLI from that file using the command:

   ```
   sfdx force:auth:sfdxurl:store -f  -s -a MyDefaultOrg
   ```

7. Finish by including a command to delete that temporary file.
8. In your CI job, you can then perform commands such as `sfdx force:apex:test:run` to, for example, run Apex tests in the default org that you authorized.

```
[illegible]$ sfdx force:org:display -u Devhub  --verbose
 ▸    sfdx-cli: update available from 6.12.0 to 6.13.0-3bc8bd73d8
Using resolved username [illegible] from alias Devhub

=== Org Description
KEY                VALUE
_________________  ____________________________________________________________
____________________________________________________________________________
____________________________________________________
Access Token       00D7F00[illegible]
YENawvX8Z8nxuSAoIK6jusD2_[illegible]
Alias              Devhub
Client Id          SalesforceDevelopmentExperience
Connected Status   Connected
Id                 00D7F00[illegible]
Instance Url       https://fighter-enterprise-6780.my.salesforce.com
Sfdx Auth Url      force://SalesforceDevelopmentExperience:1384510088588713504:5A
```

Figure 6-3. *Getting an Auth URL for use in automated processes*

auth:jwt:grant

This method is recommended by Salesforce in the Salesforce DX Developer Guide and in the Trailhead module entitled "Continuous Integration Using Salesforce DX." It is also a method useful for authorizing orgs to be used in CI context. The benefits of this method compared to the "Auth URL" approach are that

- Credentials are stored in three pieces (key, clientID, and user) instead of a single token, which some might argue is more secure.
- A custom Connected App is created for each org, and special security restrictions can be applied to that Connected App.

The disadvantages of this method are that

- Setup is more complex and time-consuming.
- The username can easily be spoofed by supplying the username of any user who is assigned to that Connected App.

This method should be used for orgs where security is of particular concern. To use this method, follow the instructions in the Salesforce DX Developer Guide.[1] We recommend that you assign only a single integration user to the custom Connected App to avoid the risk of user spoofing. Separate Connected Apps can be created for each integration.

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

Postcreation Sandbox Login

Salesforce recently introduced a new form of authentication related to sandbox cloning or refreshes. The Salesforce CLI allows you to request a new sandbox to be created, cloned, or refreshed. Sandboxes are always associated with a particular production org, which is typically also the org used as a Dev Hub for Salesforce DX scratch orgs and packages. You can initiate a sandbox clone (from production or another sandbox) using `sfdx force:org:clone`. You can then track the progress of that clone using `sfdx force:org:status`. Once the sandbox is ready, Salesforce automatically authenticates you to that org. The actual authentication is performed using an OAuth connection from the production org to the sandbox, but the tokens are passed securely to the user who requested the sandbox to be created, allowing them to immediately log in or perform actions on that sandbox.

Environment Strategy

What Salesforce orgs and sandboxes do you need to put in place to support your development, testing, and training activities?

Environment Strategy Guidelines

An effective delivery pipeline lets innovation flow from development to production. The foundation for that is a sensible environment strategy. Having multiple environments is critical for allowing development and testing to happen in an orderly way. But every extra environment increases the complexity of your delivery process, which can cause confusion, extra work, and delays. Therefore the basic guideline is to use the minimum number of orgs possible and keep the connections as simple as possible. Resist the need to add more orgs for as long as possible.

This rule does not apply to scratch orgs. Since they are temporary environments created from version control, they don't add any complexity to the development process. But the rule *does apply* to code repositories and long-running branches in those repositories. Every code repository and long-running branch increases complexity in the development workflow and increases the overhead required to keep things in sync. Your life will be simpler and happier if you can have one code repository, one main branch, and a small number of long-lived sandboxes. Branching strategy is thus related

to environment strategy, and the two need to harmonize (see Chapter 7: The Delivery Pipeline for more on branching strategy).

Multiple Production Orgs

For every production org, there needs to be an environment strategy to manage the development, testing, and training process. If you have multiple production orgs that share common functionality, there needs to be a coordinated process for delivering that shared functionality to all the orgs. Managing multiple production orgs greatly increases the complexity of the overall environment strategy.

There are generally three reasons why an organization would have multiple production orgs:

1. This was a deliberate architectural decision.
2. One company using Salesforce acquired or merged with another company using Salesforce.
3. Different business units bought Salesforce at different times without an overarching strategy.

Multiple Production Orgs As an Intentional Architecture

Why is having multiple production orgs sometimes the best architecture? Sharing a production org allows teams to share data and to share common business processes. If multiple business units do not need shared data and have entirely different business processes from one another, then it can make sense for them to have multiple production orgs. A clear example is when a holding company owns multiple different businesses (say, a book publisher and a software company) that do not interact and don't share common processes. It is almost inevitable though that a time will come when the executive leadership of the parent company might want real-time reporting from across both business units. It's also quite likely that management might want to consolidate some departments across both groups such that there later develops a need for consolidated processes between them.

It is extremely important to make the appropriate decision up front about how many production orgs to have. Greg Cook provides important guidance for the org architecture decision process in the Salesforce Developer Blog "Enterprise Architecture: Single-org

versus Multi-org Strategy."[2] I strongly suggest you study that article when making this decision, since it's extremely difficult to change later. He references this 2 x 2 matrix from the book *Enterprise Architecture as Strategy*[3] shown in Figure 6-4.

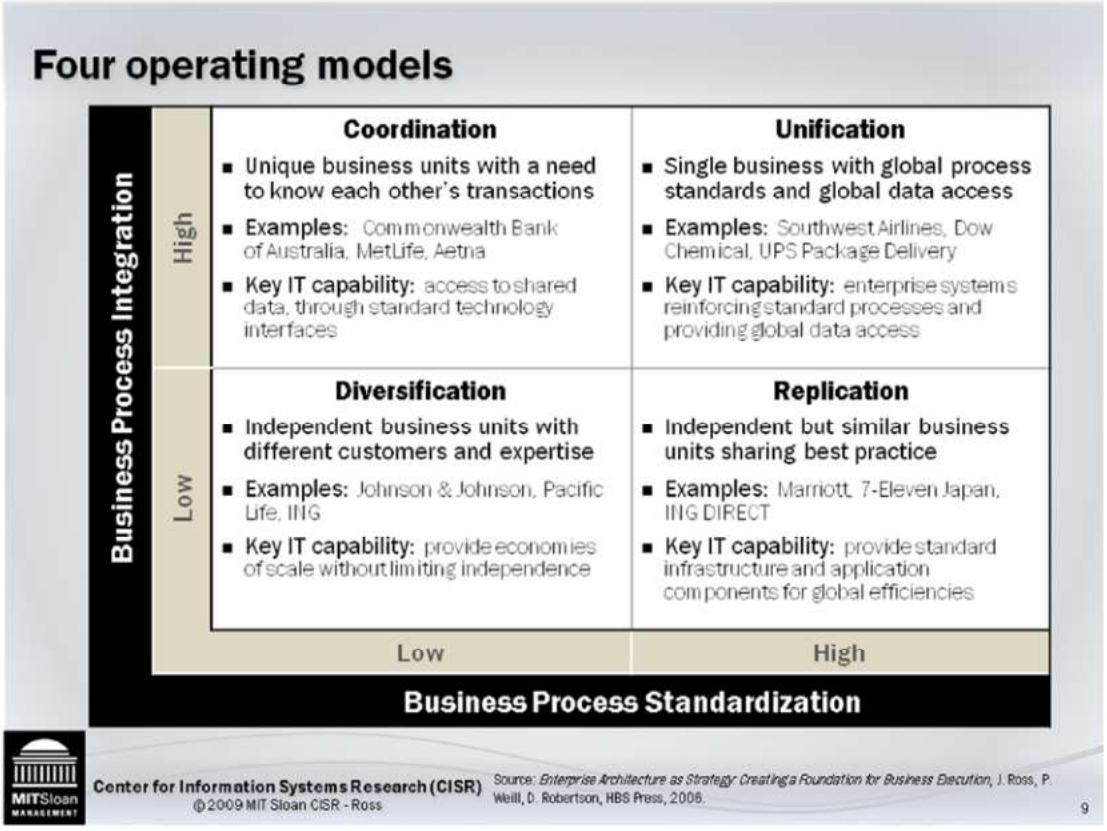

Figure 6-4. *The balance between business process integration and standardization should determine the org strategy. (Reproduced with permission from MIT Sloan CISR)*

In this diagram, high "Business Process Integration" implies the need to share common data, while high "Business Process Standardization" implies the need to share common processes by sharing Salesforce metadata. Since that blog post was first published, Salesforce has added new mechanisms for aggregating and analyzing data

[2] https://developer.salesforce.com/blogs/developer-relations/2014/10/enterprise-architecture-multi-org-strategy.html

[3] Enterprise Architecture as Strategy: Creating a Foundation for Business Execution by Jeanne W. Ross, Peter Weill, and David Robertson, Harvard Business School Press, 2006.

across orgs (Einstein Analytics), new mechanisms for sharing data (Salesforce Connect), and new mechanisms for sharing metadata (unlocked packages), which can make it much easier to manage multiple production orgs.

If independent business units continue to remain independent, with no need for shared data or processes, then they can continue to manage their environments and release strategy separately. The teams responsible for maintaining and developing their Salesforce infrastructure can and should share ideas and best practices with one another by establishing a global Center of Excellence, but there's otherwise no need for coordination.

If over time it becomes apparent that there's a need for shared data or functionality, there are several options. In the most extreme case, you can pursue an org merge (see as follows).

If the need is only for shared data reporting to be available for leadership review, you can use Salesforce's Einstein Analytics or a third-party data analytics tool to aggregate and analyze data from across multiple systems. If the need is for certain data to be available in real time to users of either Salesforce org, then you can build a custom integration between the Salesforce orgs. The easiest option for doing this is to use Salesforce Connect, which uses OData to allow Salesforce to read/write/report on live data in an external system. To share live data between orgs, you can use the Cross-Org Adapter for Salesforce Connect.

Multiple Production Orgs As a Result of Mergers or Acquisitions

It's not uncommon for one company which has one or more Salesforce orgs to acquire or merge with another company which has one or more Salesforce orgs. Integrating IT systems is a massive part of M&A, and integrating or merging Salesforce orgs is no exception. See the following for more.

Multiple Production Orgs As a Result of Department-Level Purchases

It's not uncommon, especially in large, dispersed organizations, for different departments to have purchased and to maintain separate Salesforce orgs to support their independent needs. Although Salesforce increasingly markets itself as a way to integrate all of your enterprise data, its origin is as a CRM and sales force automation tool. Different sales organizations across the company might maintain a "shadow

IT" infrastructure inside their business unit as an alternative to coordinating through corporate IT. Corporate IT may be seen as overly restrictive, requiring too long an approval process, or not having the budget or enthusiasm for taking on new projects. It's also not unheard of for Salesforce Account Executives to conspire in this approach to help them get their foot in the door with a particular customer and to gradually build a presence in an organization.

Whatever the reason, it's important to bring these initiatives out of the shadows and to establish a global Salesforce Center of Excellence to decide on the best path forward. It's in the best interest of all teams to coordinate their activities and to decide on the most efficient global architecture. Such discussions might conclude that it's beneficial for different divisions to maintain separate Salesforce orgs; the point is that this should be an intentional decision.

Merging Production Orgs

God help you if you ever need to merge production orgs. You're looking at a long, complex process that will require careful architecture and careful work. You should plan to commit significant time and resources or seek out a qualified consulting partner to help you with the process. Salesforce provides many mechanisms such as record types to segregate data and business processes, but you will need to design all of this carefully to meet the needs of the different business units.

My colleagues at Appirio have been involved in many org merges, including one multiyear org merge that resulted from a corporate merger. Shortly after the project completed, they were reengaged for a complex org split after those organizations decided to separate again!

Splitting Production Orgs

Splitting orgs is simpler than merging orgs, but is still a process that should be considered carefully (Figure 6-5). Salesforce support may be able to authorize a behind-the-scenes clone of a production org, for example, using what's known as a DOT ("Default Organization Template"). There are also third-party products, such as Sesame Software's Relational Junction, that can help with this process.

Figure 6-5. *My suggested confirmation prompt before splitting orgs*

The main point is that once an org is split, the two environments will naturally diverge from one another over time. Reversing the operation would require an org merge.

An org split requires that the metadata, data, and any integrations be migrated. Users will generally be using one org or the other, so their accounts should be deactivated on the org they are not using. If users need to access and use both orgs, this is one of many signs that you may not actually want to do an org split. Instead you can consider segregating different processes using record types or some other mechanism.

Coordinating Multiple Production Orgs

As mentioned earlier, there are new mechanisms available to coordinate data that's spread across multiple orgs and to harmonize processes that are shared across multiple orgs. Data reporting and analytics needs can be addressed by numerous business intelligence tools. Salesforce offers their proprietary Einstein Analytics tool for this purpose. This offers native integration with Salesforce, although it is limited to pulling fresh data once per hour. Salesforce has rich REST and SOAP APIs, which allow for efficient retrieval of recently modified data by specifying "If-Modified-Since" headers in the API calls. This has made it easy for most enterprise data analytics tools to build Salesforce connectors.

Salesforce Connect, and in particular the Cross-Org Adapter for Salesforce Connect, provides a mechanism to make data from one Salesforce org accessible inside another org. That data can be used in lookups, automation, Apex code, reports, and so on. The data can be read, searched, and modified. Reporting can be slow, and there are numerous limitations on that integration, but it's a much more powerful type of integration than earlier options such as Salesforce to Salesforce.

Finally, if a company has multiple production orgs (separated perhaps by geography) but wants common customizations across orgs, that's an ideal use case for unlocked packages. Unlocked packages are discussed in detail in Chapter 9: Deploying, but they allow a team to build a package of code and customization and then to syndicate it to multiple orgs. They are useful even when moving customizations to a single testing and production environment. They are infinitely more useful if you must maintain multiple production orgs. As mentioned earlier, even if orgs start with the same configuration, "configuration drift" is inevitable unless org updates are managed strictly through a continuous delivery process. When that customization is bundled into packages, it further reduces configuration drift, since a single package version is guaranteed to be consistent across all orgs.

Using second-generation managed packages is another option that is now far more accessible to enterprise developers. The original managed package process required the use of Developer orgs, Packaging Orgs, and various forms of sorcery generally only used by ISVs building for the AppExchange. Second-generation managed packages are built and function much the same as unlocked packages, although it's very important to understand their differences.

Managed packages prevent users from modifying metadata in the receiving org and obscure the contents of metadata such as Apex classes. To allow end users some degree of configurability, managed package components such as page layouts are not upgradeable. This makes managed packages less desirable in enterprises that want central control over all orgs, but more useful where an enterprise wants to provide common functionality but allow independent teams some ability to customize. In most cases, however, enterprises should be locking everybody out of the production orgs (see Chapter 12: Making It Better), so deploying unlocked packages (and updating them as needed) should provide sufficient protection against configuration drift.

Identifying and Mapping Existing Orgs

The environment strategy is the foundation for the release management process. If a team is experiencing problems with release management, there are often also problems with the environments. And environments that are outdated or out of sync are always a potential source of confusion and problems. If you happen to join an existing project and are charged with mapping and rationalizing existing environments, you can follow these tips to determine the existing structure before proceeding to make changes.

The focus here is on identifying all of the active development, testing, training, and production environments. To the degree that you find unused orgs or excessive complexity, your goal will be to simplify the environment layout by deleting those orgs (after triple-checking that they are in fact entirely unused).

The best way to determine whether an org is actually in use is by logging into that org and going to Setup ➤ Identity ➤ Login History to see whether anyone (other than you) has logged into the org. This screen provides a definitive list of every authentication against the org, including API-based read or write access. If no one has logged into the org for a long time, that indicates it's not in use by users or integrations and so is almost certainly a candidate for deletion.

By all means, ask the team if they have any existing documentation on the orgs in use. But as with everything in software development, reality trumps documentation. For each production org, you can visit the "Environments" section in the Setup UI to see a list of all sandboxes associated with that org. For each sandbox, you will see the name of the sandbox, the sandbox type, the last refreshed date, and a link that allows you to log in to that org.

If in any supporting documentation, you see indications that the team is using Developer Edition orgs, or some other type of org other than sandboxes or scratch orgs, you should view those with suspicion. Developer Edition orgs are a key part of the original managed package workflow, but otherwise should not be used as part of an enterprise development workflow. Enterprise admins have no centralized control over such environments, and they may constitute a security concern.

If the team is using a third-party release management tool like Gearset or Copado, check those systems to see what orgs are currently authorized and how they are connected to enable metadata deployments.

If the team is using change sets for their deployments, the deployment pathways are defined under "Setup ➤ Environments ➤ Deploy ➤ Deployment Settings." Each production org and their related sandboxes can be connected to each other to allow incoming change sets, outgoing change sets, or neither. To see all of the relationships, you have to log in to each org. The number of possible combinations increases geometrically based on the number of orgs. The lack of a single overview of relationships between orgs is one of many reasons why change sets don't lend themselves to fast-paced, large-scale development. In practice it's not necessary to map or control all of these relationships, just the ones leading into production. As you move to scratch orgs, you will probably also be able to delete many existing sandboxes.

If your team is using an IDE or desktop scripts to manage deployments, there's no easy way to determine all of the pathways that code and configuration may be flowing in your orgs. Establishing a clear CI/CD process will remedy that.

The purpose of building a list of active orgs and their connections is to begin to gain control over what, for many organizations, is an unruly development process. By reducing the total number of orgs that you're managing, you also greatly simplify the process of ensuring those orgs remain up to date.

Identifying Requirements for Testing, Fixes, Training, and Others

Creating a clean and rational org structure begins with determining the requirements you'll need to serve. At a minimum, you will need scratch orgs for development, one sandbox for testing integrated changes, and your production org. There are many legitimate reasons to add more environments than that, but every additional org adds complexity and requires some time to keep in sync. The benefit of adding additional orgs must be balanced against the cost of maintaining them.

The reasons for needing additional environments can be divided into these categories:

1. Developing new functionality
2. Testing functionality under development
3. Testing large volumes of data for integrations and data migrations
4. Training users
5. Resolving bugs/issues

Environment Requirements for Development

For the reasons mentioned earlier, developing new functionality should ideally be done using scratch orgs instead of sandboxes.

Despite the benefits of scratch orgs, teams who are managing massive amounts of customizations may not be able to migrate to scratch orgs in the near term. Make a plan to help your teams transition to using scratch orgs, but you may need to maintain one or more developer sandboxes during that transition.

Scratch orgs can also be used when developing custom integrations with external systems. Because scratch orgs are short-lived, their connections to external systems may have to be recreated every time a scratch org expires. It might be tempting to develop integrations such as custom Apex REST services in a sandbox that is integrated with an external test environment. Instead, practice "design by contract" by writing Apex Mocks in a scratch org to simulate requests and responses from that external system. This allows for test-driven development of those REST services and ensures robust test coverage. Then use a CI/CD process to migrate the service to an integrated sandbox for integration testing.

Environment Requirements for Testing

Testing functionality under development is itself a vast topic. Different types of testing bring different demands for test environments. This is explained in more detail in the "Test Environments" section in Chapter 8: Quality and Testing. The general strategy to support these demands is to "shift left" as much as possible and to try to batch multiple different demands into a small number of environments. "Shifting left" is a general DevOps strategy, illustrated in Figure 6-6. Visualize your development environments arranged from left to right, with the developer workstation on the far left, followed by scratch orgs, then a CI/CD server, testing sandboxes, and finally production on the far right. If a test is currently being performed in a testing sandbox, shifting that test to the left, so that it is done in a scratch org, allows the test to be performed earlier in the development process. This greatly reduces the time required to identify and resolve issues.

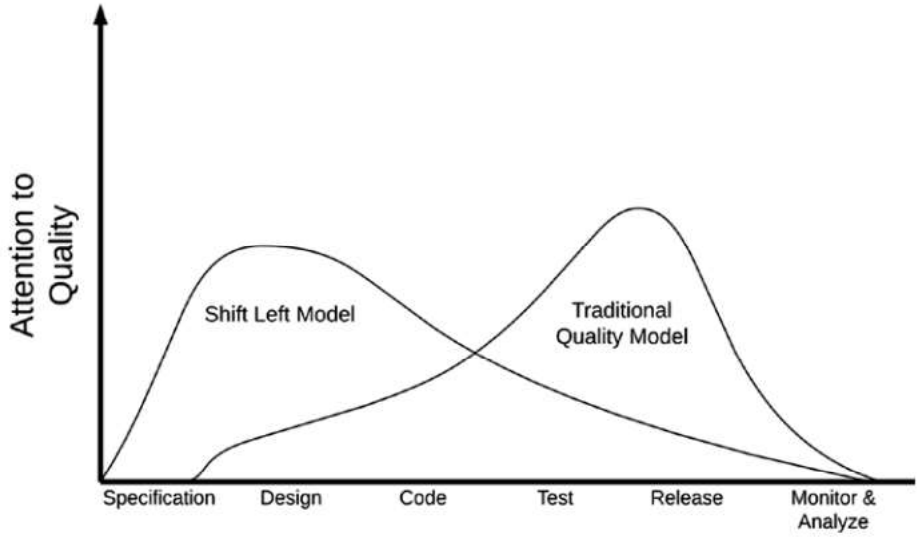

Figure 6-6. *"Shifting left" reduces overall costs by performing testing earlier in the development lifecycle*

Testing can be divided into static analysis, code reviews, unit testing, automated and manual acceptance testing, integration testing, and performance testing. Details about the environments needed for these tests are shared in Chapter 8: Quality and Testing, but these can be summarized as follows:

1. Static analysis and code reviews don't require a Salesforce org.
2. Apex unit tests and Lightning Component tests should be run both in development orgs and in other testing orgs.
3. Unit tests for JavaScript in static resources may be executable in the CI server.
4. Automated acceptance tests can be run in scratch orgs, but generally should be run in fully integrated orgs with full sample data.
5. Manual acceptance tests by the internal QA team should be run in scratch orgs; UAT should be performed in fully integrated orgs with full sample data.
6. Performance tests should be performed in a partial or full sandbox but require prior authorization from Salesforce.

Environment Requirements for Testing Integrations and Data Migrations

Salesforce integrations are ongoing connections allowing Salesforce to share data or functionality with some external system. Data migrations are one-time movements of data into or out of Salesforce. Salesforce's integration documentation explains that integrations might be performed at one of four layers, depicted in Figure 6-7.[4]

Most integrations are done at the "logic" layer, where an external system is required to get complete functionality. In this case, data is transferred between systems using APIs as part of a business process. For example, an order might be created in Salesforce and then a corresponding order created in a CPQ or ERP system. That system then sends data such as the price, order number, or order status back to Salesforce.

It's also possible to do integrations at the data layer (e.g., using External Objects), at the UI layer (e.g., using Canvas), or at the Security layer (e.g., SAML SSO).[5]

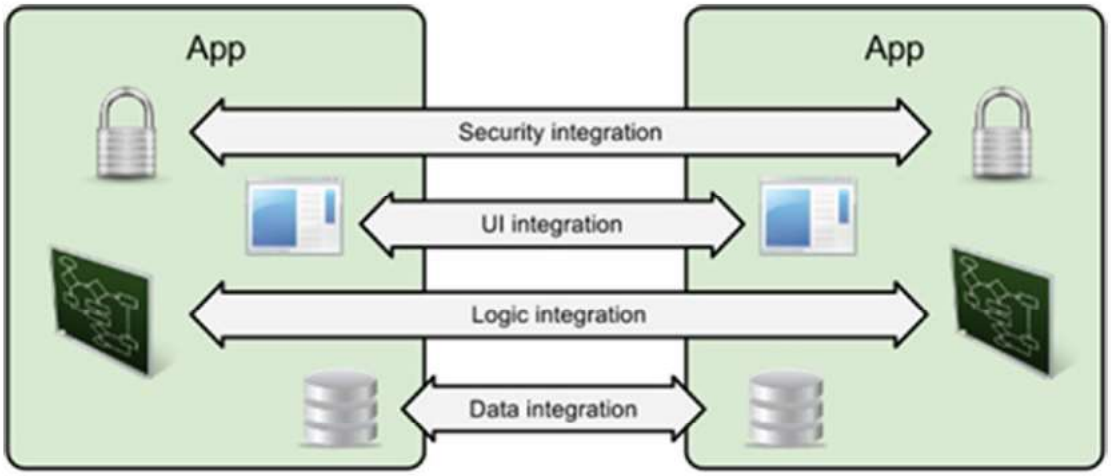

Figure 6-7. *Integration can be performed at different levels*

Can you ever use a production version of that external system for testing? If Salesforce is only reading data from that system, and there is zero possibility that the external system will be written to, and if there are no concerns about any data in the external system being exposed to someone who shouldn't have access, then it might be

[4]https://developer.salesforce.com/page/Integration

[5]https://developer.salesforce.com/blogs/developer-relations/2015/06/integrating-force-com.html

acceptable to test the integration against a production system. But that's rarely the case unless you're integrating with a public read-only service such as Google Maps or the US Postal Service's address validation service. In all other cases, these integrations require a test instance of the external system.

Not all platforms are as generous as Salesforce in providing sandbox or test environments. This means that the availability of test instances of external systems might be a limiting factor in what Salesforce environments you can support. If you're building a two-way integration with a product that has only three instances—development, test, and production—then you can only have three Salesforce environments integrated with this system. If you want the development instance to be available to your developers, then you have only one nonproduction instance left to use. You'll have to choose whether to make that available to your UAT testing environment, to your training environment, or to some other environment.

Environments where there are multiple external systems integrating will generally benefit from having a middleware tool in place. In those instances also, there should be development, testing, and production instances of the middleware, each connecting the appropriate environments.

For any kind of large-scale data import into Salesforce, you will almost certainly need a separate partial or full sandbox to test in. Scratch orgs and Developer sandboxes have extremely limited data volumes (200 MB). Developer Pro sandboxes allow 1 GB, Partial Copy sandboxes allow 5 GB, and full sandboxes give the same storage amount as the production org. Most Salesforce records require 2 kB of storage space, which means that a Partial Copy sandbox can store 2.5 million records. If your data migration is larger than that, you'll need a Full Copy sandbox, which might require that your organization buy an additional one.

Like software development, data migration is an iterative process that will likely require multiple attempts. Your data team will need the freedom to delete and reimport data multiple times before they have satisfactorily resolved all issues with character encoding (hint: use UTF-8), field length, relationships, picklist values, and so on. Do everyone a favor and put them in their own environment.

For small-scale testing of integrations or small migrations of new data, you might be able to safely reuse your UAT sandbox for this purpose. That allows the UAT environment to be fully integrated, available for end user testing, and also available for the integration team to validate their work. But this approach requires care, and you should make a clear segregation between the records that the integration team is touching and the records being used by your UAT testers.

Environment Requirements for Training

To support employee onboarding or when training your employees in new processes, it's customary to provide a dedicated training environment. In some cases, this can be the same as the UAT testing environment. If you can clearly demarcate training data from business data, you might also be able to do training in production. Even when you're rolling out new functionality, in some cases you can use feature flags (see Chapter 10: Releasing to Users) to deploy that functionality to production. Record types can be used to enable distinct layouts or business logic, and so users can be trained on forthcoming features by providing them training data that uses a new record type that exhibits different behavior. When you're ready to roll out the new processes on all data, just change the record types of your existing data.

If none of these scenarios are suitable, you may need a dedicated environment for training. You should include this training environment in your delivery pipeline so that its metadata is automatically updated at the same time as your other environments. This reduces the burden involved in maintaining a separate environment.

Environment Requirements for Resolving Bugs/Issues

Most bugs discovered in production are minor enough that they can be logged and fixed by the development team as part of their normal release process. However issues that cause downtime or data corruption require immediate fixes.

Your goal should be to handle even critical production hotfixes through your normal delivery pipeline without creating a separate process for this. The speed with which the development team can restore production systems and the speed with which new updates can be deployed are two of the five key metrics for measuring your software delivery performance (see Chapter 3: DevOps).

The practice of continuous delivery means that the master branch of your repository is always able to be released to production. If you've achieved this, then your normal, linear delivery pipeline will also be the conduit through which emergency bug fixes can be deployed. Developers will always be working on half-baked functionality, but that functionality should be hidden behind Feature Flags such that it can be safely deployed without releasing.

If your release process is not yet at this point, it may be tempting to bypass your CI/CD pipeline entirely in an emergency. This is inevitably a bad idea, since it means that you are making changes in a panicked state of mind *without tracking those changes*

in version control. There are many horror stories of teams making problems worse by implementing hasty and untracked changes.

Some of the most difficult-to-debug production issues involve data issues. Since full sandboxes can only be refreshed every 30 days, it's not always easy to debug these issues outside of production. But accessing debug logs requires "View All Data" permission, which is not something that developers should have in production. To address this, you can consider having a separate hotfix environment.

Teams doing periodic major releases often maintain a partial sandbox as a separate hotfix environment. Immediately after the production deployment, they refresh this sandbox to ensure its metadata and at least a subset of its data are in sync with production. This environment can be connected to your CI/CD system so that changes made to a particular branch (e.g., "hotfix") will be deployed to this environment. If there is a problem, you can create a hotfix branch off of the master/production branch and push urgent changes to this environment for testing. If the fix is successful, it can then be put into the delivery pipeline so that all environments (including production) get updated.

Creating and Using Scratch Orgs

The short-lived nature of scratch orgs makes them an ideal environment for development, demos, and most types of testing. How do you create and use scratch orgs for these purposes?

General Workflow

The main body of your Salesforce metadata should ideally be built in scratch orgs, published as unlocked packages, and installed and kept up to date in testing and production environments. Metadata that cannot or should not live inside a package should be managed at the org level.

Salesforce DX introduces the concept of unlocked packages. Unlocked packages allow related functionality to be compiled and deployed together as a single unit. Packages are versioned, and the Setup UI shows what metadata is in a particular package, in which version of the package it was added (or deprecated), and which package each piece of metadata belongs to. Importantly, packages also allow for metadata deletions to be safely propagated from development to production.

Thus unlocked packages are generally the best method of moving features from development to sandboxes and production orgs.

Some types of metadata, like org Settings and Profiles, apply to an entire org and not to any one package. They therefore need to be managed using the metadata deploy and retrieve commands. The main reason certain types of metadata cannot be included in packages is that it could lead to situations where multiple packages have conflicting configuration, with no way to discern which should take precedence.

Packages cannot include Setting metadata. But you can specify the Settings that are required in the target org by specifying a `definitionFile` for that package. The definitionFile is a scratch org definition file that specifies features and settings that should be present in an org. This file is used in the background during the package version creation process. It also allows you to explicitly state the environment dependencies for every package rather than just implicitly depending on certain features or settings. This explicit definition makes it very clear how orgs must be configured before receiving this package.

Grouping metadata into folders allows you to logically organize your metadata, which is a capability that was sorely lacking before DX. Depending on the needs of your project, you can store multiple packages in one repository, although in most cases it will be easier to automate your workflow if each package is in its own repository. The source "synchronization" capability of scratch orgs means that developers can easily sync changes from the org's Setup UI to their local machines without having to be experts in deciphering the different types of metadata. Once downloaded, that metadata can be viewed, edited, and tracked in version control in the developer's IDE.

Tracking changes using version control allows developers to easily merge their work with others on their team, so that scratch orgs can be independent of one another but also easily synchronized with recent changes from others.

Building packages ensures that metadata is guaranteed to be identical across orgs, and means that deployment errors can be resolved during the package publishing process as opposed to recurring every time the same metadata is deployed to different environments.

Managing org-level metadata (along with which version of different packages is installed) in a central org-level repository allows central control and visibility of org-level configuration and differences. Org configuration can be identical across orgs where appropriate, and can also vary if needed.

Creating Scratch Orgs

Scratch orgs are created by sending an org creation request to a Dev Hub. The creation request must reference a scratch org definition file, which determines the initial characteristics of the org. Creation requests are typically made using the Salesforce CLI, although Salesforce's VS Code extension and an increasing number of third-party tools provide wrappers around that process.

Since the purpose of scratch orgs is to build new functionality destined for a production org, you need to use the scratch org definition file (such as shown in Listing 6-1) to match the characteristics of the target org. Applications always depend on certain underlying functionality in their Salesforce org; the scratch org definition file makes those dependencies explicit. Those developing managed packages will likely need to test using multiple scratch org definition files to simulate a variety of different installation orgs. Those developing for a particular business may only need a single scratch org definition file, although if that business has multiple production orgs, or is experimenting with enabling certain features, they too may need multiple scratch org definition files. Don't worry, they're short and easy to create.

Listing 6-1. Sample scratch org definition file

```
{
  "orgName": "Acme",
  "edition": "Enterprise",
  "features": ["Communities", "ServiceCloud", "Chatbot"],
  "settings": {
      "orgPreferenceSettings": {
          "networksEnabled": true,
          "s1DesktopEnabled": true,
          "s1EncryptedStoragePref2": false
      },
      "omniChannelSettings": {
          "enableOmniChannel": true
      },
```

```
        "caseSettings": {
            "systemUserEmail": "support@acme.com"
        }
    }
  }
```

There are a number of optional configurations in a scratch org definition file, such as the `orgName`. But the most important configurations to understand are `edition`, `features`, and `settings`. There are also two other significant configuration options that as of Spring 2019 are still in pilot: `sourceOrg` and snapshot.

The `edition` option corresponds to the Salesforce edition, explained earlier in the section "Different Types of Orgs," and simply allows the user to select "Developer," "Enterprise," "Group," or "Professional." Unlimited edition orgs are just Enterprise edition orgs with higher limits. Match the edition to your target org. For example, Developer edition orgs have many features enabled that would otherwise only be available as paid add-ons. If you build a feature based on Work.com capabilities in a Developer Edition org, don't expect that to work in an Enterprise org that does not have Work.com installed.

The `features` option generally corresponds to paid features that are not available in every org, whereas the `settings` option corresponds to org configuration that is available for free, but may need to be enabled in an org. For example, the `SalesWave` feature can't just be turned on in any Salesforce org … that's a paid capability that would normally require your organization to have purchased Einstein Analytics. Fortunately for developers, you **can** create a scratch org that has Sales Wave enabled, so you can build apps on top of those capabilities. As an aside, there is often a disconnect between the backend/developer names for things in Salesforce and their current marketing names. Sales Wave is now called "Einstein Analytics for Sales Cloud Einstein," but the older name `SalesWave` remains on the backend to ensure that developer-built integrations don't break.

The `settings` configuration allows you to specify any setting that is available in the Metadata API (see https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_settings.htm). These are configuration options that can be specified through the setup UI.

When you are initially migrating from org-based development to source-based development, expect to spend some time adjusting the features and settings in your scratch org definition file until you can get your metadata deploying successfully to your

new scratch org. For example, I spent several hours trying to diagnose why I was getting `Variable does not exist: Suffix` errors when trying to push metadata to a scratch org. Eventually, I found that there is a `nameSettings` metadata item that allowed me to specify `enableNameSuffix`.

Although this kind of debugging can be tedious, for those familiar with Salesforce release management, this will feel familiar. Identifying missing dependencies or differences in org configuration is one of the most common challenges in Salesforce release management. The massive change that Salesforce DX brings, however, is that instead of having to repeatedly diagnose and resolve obscure deployment errors every time you move metadata to a new org, those errors can be resolved up front while defining your scratch org configuration and getting your packages to publish. Once those dependencies are made explicit, it is straightforward to confirm that the necessary features and settings are in place before attempting to install a package.

The `sourceOrg` and `snapshot` configuration options relate to two features that are still in Pilot as of this writing: Org Shape and Scratch Org Snapshots. Although there is an enormous amount of configuration that can be defined through features and settings, there are some kinds of configuration like org-wide email addresses that have lingered for many years without an easy way to automate their setup. Gaps in the Metadata API mean that in some cases a scratch org can't be completely defined from source and that manual steps are required to complete setup.

Salesforce has responded to that gap in several ways. First, they've now put in place an automated process to ensure that any new features are fully covered by the Metadata API. The Metadata Coverage Report (`https://developer.salesforce.com/docs/metadata-coverage/`) is dynamically generated from that automated process. Second, they continue to work with the engineering teams responsible for different parts of the platform to help them ensure that their configuration can be controlled using the Metadata API. Finally, to bypass those gaps, Org Shape and Scratch Org Snapshots allow scratch orgs to be created based on templates, instead of being based on an Edition.

Org Shape and Scratch Org Snapshots

Certain features of a production org are not yet available in scratch org definition files. Similarly, for some production orgs, Salesforce may have approved increased limits (number of custom objects, lines of Apex code, etc.) compared to what is standard for a particular edition. And there are also many settings that can't be represented in the Metadata API and thus are not supported in scratch org definition files. Thus it may not

be possible to replicate some characteristics of production orgs using just scratch org definition files.

Org Shape creates a template based on a particular production org that can be used to create a scratch org. The org shape mimics the features, limits, and settings of your production org without the extraneous data and metadata. This means that org shape does not contain data or metadata, but is otherwise identical to the characteristics of a production org. This allows scratch orgs to be created with features or settings that cannot be represented in scratch org definition files. When creating a scratch org based on org shape, a reference to the shape ID is used in place of specifying an `edition` in the scratch org definition file. Note that org shape is not automatically updated when the production org's settings are updated; you'll need to recreate the shape.

By contrast, Scratch Org Snapshots create templates based on the configuration of a particular scratch org. You first create a scratch org, then manually adjust its settings, then capture a snapshot which you can then use to create other scratch orgs. Just like with org shape, when creating a scratch org based on a snapshot, a reference to the snapshot is used in place of specifying an `edition` in the scratch org definition file.

Scratch org snapshots work similar to org shape, with some very nice additional characteristics. First of all, you can create a scratch org based on org shape and then capture a snapshot of that! This allows you to begin by capturing the characteristics of your production (target) org and then modify those as needed and use that as the foundation for your development and testing environments. Further, you can also install managed packages and other metadata in the scratch org, add data for testing and configuration, and capture a snapshot that includes all of that customization.

The installation of dependent packages is one of the most time-consuming aspects of scratch org initialization. Large managed packages such as Vlocity can take up to 30 minutes to install, making scratch org setup time-consuming and burdensome. The forthcoming Scratch Org Snapshots feature will help address this issue. A scratch org can be created and configured, and have managed packages and other metadata loaded into it. You can then take a snapshot of that org and use that snapshot as the template for creating new scratch orgs. Scratch orgs created in this fashion have the managed packages in place, thus greatly speeding the setup process.

Table 6-1 summarizes which aspects of an org are (and are not) carried over to different representations of an org.

Table 6-1. How an org is created determines which aspects of the org are populated

Org Aspect	Full Sandbox Copy	Partial Sandbox Copy	Developer Sandbox Copy	Scratch Org Snapshot	Scratch Org Shape	Scratch Org Definition File
Can be the basis for Scratch Orgs				✓	✓	✓
Can be the basis for packages				✓		✓
Edition	✓	✓	✓	✓	✓	✓
Features	✓	✓	✓	✓	✓	✓
Org limit overrides	✓	✓	✓	✓	✓	Partial
Settings supported by the Metadata API	✓	✓	✓	✓	✓	✓
Settings not in the Metadata API	✓	✓	✓	✓	✓	
Managed packages	✓	✓	✓	✓		
Metadata	✓	✓	✓	✓		
Partial data	✓	✓				
Full data	✓					

Although org shape and snapshots are extremely useful, they are not a substitute for capturing and deploying configuration from version control. To the degree possible, the settings and packages in a snapshot should be specified in your configuration files, and you should automate the build process for your snapshot. Thus you can use the snapshot to speed up scratch org creation while still having confidence that you know the characteristics of that snapshot and how they evolve over time.

Using either org shape or snapshots makes the underlying configuration opaque, in that you are not explicitly stating the features and settings that a package requires. But when defining packages and orgs based on these templates, you can also explicitly specify features and settings that you require. This allows you to use the org template to address gaps in the Metadata API while still being as explicit as possible about the required org configuration. Tracking every aspect of org configuration in version control provides traceability and makes it easy to move those packages to other orgs in the future.

Scratch org snapshots are extremely powerful. But you should be careful about what you include and do not include in the snapshot itself. Your snapshot should contain only the underlying dependencies that your project is building upon, not any of the metadata that your team may need to update. For example, they can and should contain third-party managed packages and metadata that is common to other teams and not under active development. The idea is to include all the foundations for your team's own customizations, but not to include the actual metadata for your project. Instead, push that metadata from version control to the scratch org you create from the snapshot. That ensures that if you need to delete or modify any of that metadata, your version control system remains the source of truth.

Initializing Scratch Orgs

When a scratch org is first created, it has an edition, features, settings, and one dynamically created admin user. To proceed with developing on scratch orgs, there are several other steps that are required. In most cases, those setup steps are

1. Install dependent packages
2. Push metadata
3. Create additional user accounts
4. Load sample or testing data
5. Run additional setup scripts

Install Dependent Packages

Whereas the scratch org definition file defines the scratch org's characteristics, Salesforce DX projects also contain an `sfdx-project.json` file that defines your package configuration. Listing 6-2 shows an example of this file. Salesforce DX project metadata is grouped into folders called "packages" based on the expectation that this metadata will be built into individual packages, versioned and installed in target orgs.

The most common initial step for initializing a scratch org is to install other package dependencies. You may be making use of functionality tied to the Marketing Cloud Connector, in which case you'll need to install the latest version of that managed package. The `sfdx-project.json` file allows each package to define other packages on which they depend. These package dependencies are specified using an alias, with the alias defined in one place in the file. For external package dependencies, you specify the alias using the ID of a particular package version. If one package in your project depends on other packages in the same project, those dependencies should also be specified, but in this case they can reference the package by name and specify a `versionNumber` for that package.

Listing 6-2. Sample `sfdx-project.json` configuration file

```
{
   "namespace": "",
   "sfdcLoginUrl": "https://login.salesforce.com",
   "sourceApiVersion": "43.0",
   "packageDirectories": [
      {
         "path": "util",
         "default": true,
         "package": "Expense Manager - Util",
         "versionName": "Spring '18",
         "versionDescription": "Welcome to Spring 2018 Release of Expense
         Manager Util Package",
         "versionNumber": "4.7.0.NEXT",
         "definitionFile": "config/scratch-org-def.json"
      },
```

```
        {
           "path": "exp-core",
           "default": false,
           "package": "Expense Manager",
           "versionName": "v 3.2",
           "versionDescription": "Spring 2018 Release",
           "versionNumber": "3.2.0.NEXT",
           "definitionFile": "config/scratch-org-def.json",
           "dependencies": [
              {
                 "package": "Expense Manager - Util",
                 "versionNumber": "4.7.0.LATEST"
              },
              {
                 "package" : "External Apex Library - 1.0.0.4"
              }
           ]
        }
     ],
     "packageAliases": {
        "Expense Manager - Util": "0HoB00000004CFpKAM",
        "External Apex Library - 1.0.0.4": "04tB0000000IB1EIAW",
        "Expense Manager": "0HoB00000004CFuKAM"}
  }
```

Although this is a highly requested feature, as of this writing, Salesforce does not automatically install dependent packages for you as part of scratch org setup. You can write scripts to automate the scratch org setup process or use third-party tools such as Appirio DX, the Texei plugin,[6] or the Rootstock dependency utility[7] to manage this.

As mentioned previously, to speed the scratch org creation process, you can use a scratch org snapshot that has these managed packages already installed.

[6] https://github.com/texei/texei-sfdx-plugin

[7] www.npmjs.com/package/rstk-sfdx-package-utils

Push Metadata

After installing packages, your typical next step will be to push the metadata for your packages to the scratch org. Your local metadata typically depends on other packages, whereas user permissions and data depend on your metadata; thus the order of these stages is important.

The Salesforce DX command for pushing your source metadata to a scratch org is `sfdx force:source:push`. The command allows you to specify which scratch org you want to push to, but does not allow you to limit the metadata that you are pushing.

Pushing metadata and installing dependent packages both take time. If you're dealing with a small org, the process may only take a few minutes. But as your org's complexity grows, the initial setup time for a scratch org will grow as well. If you find that your scratch org creation is taking more than 10 minutes, you might consider mechanisms to initialize your scratch org with only the subset of functionality relevant to your current work. The larger your team and more complex your org, the more likely you are to need a selective process for initializing these orgs.

The simplest method for narrowing the scope of metadata in a scratch org is to divide your codebase into several repositories. If you have two relatively independent teams developing functionality for your org (e.g., sales cloud vs. service cloud), you might have three code repositories, one for each team, and one for shared functionality. The repository governing shared functionality can be built as a package that is then installed as an external dependency for the two team-specific repositories. Each team repository otherwise stands alone, allowing the creation of a scratch org with just the functionality needed for that team. Even when code is divided into multiple repositories, you can easily work across multiple folders by adding them to a single workspace in your IDE. Visual Studio Code and most other IDEs allow multiproject workspaces, although not all extensions fully support this.

It's worth noting that if you want to limit the metadata that is pushed to a scratch org or ignore non-metadata files present in the same folders, you can use a `.forceignore` file like that shown in Listing 6-3. The `.forceignore` file has a syntax similar to the `.gitignore` file used with Git. Metadata files or folders listed in a `.forceignore` file will not be pushed or pulled from that scratch org.

Listing 6-3. Sample `.forceignore` file

```
# Specify a relative path to a directory from the project root
helloWorld/main/default/classes

# Specify a wildcard directory - any directory named "classes" is
excluded
classes

# Specify file extensions
.cls
.pdf

# Specify a specific file
helloWorld/main/default/HelloWorld.cls
```

Create Additional User Accounts

Pushing metadata to the scratch org establishes the basic capabilities of that org. But Salesforce's flexible permission system means that different users can have substantially different views and access, even in the same org. Proper testing requires that you impersonate the user permissions of the users you are developing for, and to do this you may want to create additional user accounts in your scratch org.

The Salesforce CLI provides the `sfdx force:user:create` command for creating users. When running this command, you can add a `--setalias` parameter to create a short name that can be used to refer to that user in subsequent commands.

Similar to the previously mentioned SFDX configuration files, Salesforce also makes available a user definition file such as the one shown in Listing 6-4 that can be used to create users that fit standard personas. You specify this using a `--definitionfile` parameter.

Listing 6-4. Sample user definition file

```
{
    "Username": "tester1@sfdx.org",
    "LastName": "Hobbs",
    "Email": "tester1@sfdx.org",
    "Alias": "tester1",
    "TimeZoneSidKey": "America/Denver",
```

```
    "LocaleSidKey": "en_US",
    "EmailEncodingKey": "UTF-8",
    "LanguageLocaleKey": "en_US",
    "profileName": "Standard Platform User",
    "permsets": ["Dreamhouse", "Cloudhouse"],
    "generatePassword": true
}
```

You will typically want to omit the "Username" property and allow Salesforce to autogenerate this, since it needs to be unique across all Salesforce orgs. The other properties in this file, especially the `profileName` and `permsets`, allow you to specify the permissions assigned to this user, thus defining their level of access and how the org behaves for them.

As you develop features in service of your users, spend time refining these user definition files, and make them available to your developers and testers so that they can easily view the org from the point of view of these sample users.

Load Sample or Testing Data

One major difference between scratch orgs and sandboxes is that scratch orgs contain no metadata or data from a production org. Scratch org definition files have a property `hasSampleData` that can be used to populate the scratch org with data similar to that in developer edition orgs. But you will want to curate appropriate sets of sample data for your organization.

The Salesforce CLI provides several data-related commands for exporting and importing modest amounts of data for development and testing.

`sfdx force:data:tree` commands should be your first approach. These commands handle the challenge of creating related records in a single step, without the need to explicitly specify external IDs. If you wanted to manually load some related Accounts, Contacts, and Cases, you would first need to create the Accounts, get their IDs, and then create the Contacts, specifying their Account IDs, and then create the Cases, specifying their Contact IDs. `sfdx force:data:tree` allows you to export and import a JSON file that expresses hierarchical data without using IDs. The data tree API processes these files and handles the process for you.

To use this command, generate appropriate testing data in an org, and then run the sfdx `force:data:tree:export` command, specifying a SOQL query to export the data. By using subqueries like `sfdx force:data:tree:export -p -q "SELECT Id, Name,`

`(SELECT Id, LastName, FirstName FROM Contacts) FROM Account"`, you can export data, including multiple levels of relationships. The result of this query is one JSON file for each object and another JSON file to specify the relationship between each object. Although IDs are specified in the query, they are ignored when creating this data tree so that the records can be inserted into any org.

Once you've exported that data, you can then import it using the `sfdx force:data:tree:import` command, which takes the generated files as input and inserts corresponding records, establishing relationships between them.

As your org evolves, plan to manage this sample data alongside the code, regularly updating it to capture the diversity of objects and data you are working with. This will never substitute for testing in a staging sandbox against a full volume of production-like data, but it allows you to provide realistic data to your developers and testers and to add special cases as they are discovered.

The `sfdx force:data:tree` commands are extremely convenient but are limited to importing 200 records at a time. If you need to import larger volumes of data, you can use commands such as `sfdx force:data:bulk` which is a complete wrapper around the Bulk API. The Bulk API is the mechanism used for importing or exporting large volumes of data (thousands or millions of records).

The Salesforce CLI provides an additional set of commands `sfdx force:data:record` for modifying individual records. If appropriate, those can be added to setup scripts to fine-tune data as needed.

One important use of data loading is to populate configuration data that is not stored in custom metadata. See "Configuration Data Management" in Chapter 4: Developing on Salesforce.

Run Additional Setup Scripts

Even all of the preceding steps are sometimes not sufficient to complete the org setup process. As the icing on the cake, you can run additional scripts to complete the setup. The most common kinds of scripts are anonymous Apex and Selenium UI automation scripts.

Anonymous Apex are blocks of Apex code that are executed in an org but not stored in that org. They can be run from the developer console, but can also be run using the Salesforce CLI's `sfdx force:apex:execute` command. Any kind of setup scripts should be stored in your code repository along with your code. You can then use that command by specifying the Apex code file and the user to execute as. Anonymous Apex

is particularly useful if you want to query a group of records and do some simple batch processing on them.

Selenium is the most common type of UI automation, although there are other types such as Tosca and Provar. Selenium is most commonly used for UI testing, but can also be used to script activities that can only be done through the UI. Org shape and scratch org snapshots may be able to address limitations in the scratch org setup process by capturing settings that are not supported by the Metadata API and allowing you to create scratch orgs based on manually configured orgs. But until those are generally available, you can use Selenium automation for this purpose.

Selenium scripts use the web browser to simulate user actions such as logging into an org, navigating to a particular page, and changing a setting. They require care to craft in such a way that they continue to work even as Salesforce rolls out updates. UI automation is notoriously brittle, so you will do well to get help from someone who is experienced with building resilient Selenium scripts for Salesforce testing. As with anonymous Apex, these scripts should be stored in your code repository to ensure they are versioned and can be updated along with your source code and configuration as your org evolves.

Developing on Scratch Orgs

Scratch orgs provide two main characteristics that make them ideal for development. First, because they can be created entirely from configuration stored in version control, they behave and can be recreated in a predictable way. In other words, they are always in a "known" state, unlike long-lived sandboxes which might be modified by other users in ways that have unpredictable impact. Second, they allow for simple "source synchronization" that can synchronize the org configuration with your local file system (and thus with version control) in a single step. Source synchronization allows the state of the application to evolve in a controlled way that is tracked in version control.

For developers who are accustomed to working in long-lived developer sandboxes, this means that their development workflow has at least one additional step: they need to periodically recreate a development scratch org. Whether developers create development scratch orgs many times per day, once per week, or once per month (the maximum lifespan of a scratch org) depends on several factors, outlined here.

First, do you need different types of scratch orgs to work on different applications? If your org is complex enough to require different setup paths for different applications,

then developers may need to create multiple scratch orgs to work on those different applications. They may find themselves recreating these orgs as they move between different work items.

Second, how long does your scratch org take to create, and have you completely automated the setup? Until capabilities such as scratch org snapshots are generally available, teams may find that it takes tens of minutes or up to an hour or more for automated package installations to complete. If on top of that there are manual configuration steps that have to be performed, this will add to the setup time and make recreating orgs regularly an inefficient process. This challenge should push teams to look at additional automation such as Selenium UI automation, but may also decrease the frequency that orgs are created.

Finally, if developers are using a feature branch workflow, where features are developed in separate version control branches which each require review and approval, this leads to the need to create multiple independent scratch orgs, one per branch, to ensure that each feature branch is truly distinct from each other. For teams that are just getting started with using version control, and need to ensure rigorous testing and approval of each feature in isolation, a feature branch workflow can be helpful. However it is not as efficient as trunk-based development (continuous integration) and tends to limit refactoring and collaboration and lead to increased merge conflicts.

Once the developer creates the scratch org, they can then make code changes on their local machine and run an `sfdx force:source:push` to push them to the scratch org. As they make changes in the org itself, they can run `sfdx force:source:pull` to pull those configuration changes into their local codebase. Salesforce's extensions for VS Code recently introduced a compile on save functionality that will automatically push local source to the scratch org when files are updated. This is particularly helpful when pulling or merging changes from shared version control, since this triggers a push when the local source is updated. A variation on this is the capability included in Appirio DX which watches the scratch org for changes and periodically runs synchronization.

Scratch orgs are generally intended to be worked on by one developer at a time. But there are some cases where collaborating in a scratch org can be useful. If the click-based developers on your team don't have an easy way to create scratch orgs and capture changes in version control, they may need to partner with code-based developers who can provision such an environment for them and capture their changes in version control. This can be a good way to force collaboration within the team and for these two developers to work together using their complementary skillset.

Development in a scratch org is otherwise identical to development in any other Salesforce environment. Other sections of this book such as Part 2: Salesforce Dev provide an overview of how to approach development with the whole dev lifecycle in mind.

Scratch Orgs As Review Apps

In addition to their use as development environments, scratch orgs are well suited to be used as review apps. Review apps are environments created dynamically from version control that provide an isolated environment to review features that may still be under development. The concept was popularized by Heroku.

A review app is simply a scratch org that is created for testing or evaluation purposes and is not intended for development. As such, they are perfect for performing quality analysis, functional review, or demos.

An important characteristic of a review app is that they should be self-service. A tester, reviewer, or person giving a demo should be able to create a review app for themselves with minimal difficulty and without relying on developers for assistance. If you are not using a commercial tool, you can build such a capability using a manual job in your CI system that runs the same setup script that your developers use to create their development environment. After initializing the org, generate a one-click login URL by running `sfdx force:org:open --urlonly`. Reviewers can execute that CI job to initialize an org, which they can open directly using that login URL.

Cloning and Refreshing Sandboxes

For most of Salesforce's history, there has been only one path for creating or "refreshing" a development or testing environment: creating a sandbox that is a copy of the production environment. This greatly limited the possible ways that sandboxes could be managed.

Fortunately, with the arrival of Salesforce DX, there are now a diversity of org creation and refresh options. For example, it's recently become possible to clone one sandbox from another, and Lightning Dev Pro sandboxes will make it possible to create and destroy sandboxes dynamically as needed. Having discussed scratch orgs, we can now look at the options for sandboxes and how they can be used beneficially.

Sandboxes are managed from the Sandboxes section in the Setup UI in your production org. That screen shows a list of your existing sandboxes, along with options to create, clone, refresh, and delete sandboxes.

Creating, Cloning, or Refreshing Sandboxes

In essence, creating, cloning, and refreshing sandboxes all do the same thing. All of these actions replace the sandbox with a new copy based on a source org. If a new sandbox is being created from production, this is called "creating a sandbox." If a new sandbox is being created from another sandbox, this is called "cloning a sandbox." And if an existing sandbox is being refreshed, either from production or from another sandbox, this is called "refreshing the sandbox."

Refreshing a sandbox is equivalent to destroying that sandbox and recreating it. The only thing that is preserved during a sandbox refresh is the name of that sandbox and its type (Developer, etc.).

In all of these cases, the features, limits, settings, and metadata of the source org are recreated in the target org. To what extent **data** is also copied depends on the situation.

Cloning or refreshing a sandbox from another sandbox requires that both sandboxes be of the same type, and a complete copy of the source sandbox's data is made into the target sandbox. If you are creating or refreshing a sandbox from **production**, data may or may not be copied, depending on the type of your target sandbox.

Full copy sandboxes receive a complete copy of the production org's data. Creating, cloning, or refreshing full copy sandboxes can take a very long time (a day or two) if the source org contains large volumes of data. Salesforce also limits these updates so that the org can only be refreshed once every 29 days.

Partial copy sandboxes created from production receive a subset of production data, based on a filter called a sandbox template. The sandbox template is used to determine which objects' data will be copied and how many records. This allows for a faster creation and refresh process, and partial copy sandboxes can be refreshed as often as once every 5 days.

Developer and Developer Pro sandboxes created from production do not receive any data. Their creation is generally quite fast, and they can be refreshed as often as once per day.

Because the sandbox refresh process is a destructive process, Salesforce requires that newly refreshed orgs be activated. In reality, the newly refreshed org is a different org

than the sandbox being refreshed. Activating simply deletes the existing sandbox and redirects the sandbox name to the newly created org. Activation thus gives the ability for teams to continue working in an existing sandbox until the refreshed version is available, so that they can save their work and then resume working in the newly activated org immediately.

Salesforce provides the ability to perform setup actions by triggering the execution of an Apex class after a sandbox is created or refreshed. This setup script can perform a variety of actions. Common scenarios include deleting or obfuscating sensitive data in the org, creating or updating users, and so on.

There are a number of AppExchange products such as Odaseva and OwnBackup that provide helpful capabilities such as copying or sanitizing large volumes of data when creating testing environments.

Planning Org Refreshes

Understanding the org refresh process enables you to make an appropriate plan to support your development and testing needs.

The function of org refreshes is to bring metadata and data into sync between environments. Historically, org refreshes have been one of the main ways that teams have attempted to create consistent metadata between orgs. But configuration drift (metadata differences) between orgs is a symptom of lacking governance and continuous delivery. So you should use the mechanisms described in this book to ensure metadata stays in sync and rely on sandbox refreshes mostly for syncing data.

Making small, frequent deployments reduces the risk of any single deployment into production. Building your metadata into unlocked packages reduces differences between orgs and simplifies the deployment process. Locking everybody out from changing metadata in sandboxes and production ensures that orgs behave consistently and that changes are tracked. These are the mechanisms you should rely on to ensure that metadata is consistent. Note that these methods lead to changes being propagated from development through testing to production. The old pattern of refreshing sandbox metadata from production implies a lack of control over the deployment process, with changes in production being back-propagated to development and testing environments instead of the other way around.

But org refreshes still provide a clear and simple way to update testing environments with production-like data. Your final UAT or staging environment should closely resemble your production environment. One reason for this is to ensure that functionality is performant in the face of large data volumes. Many Salesforce solutions have been sent back to developers after testing in a full sandbox showed unacceptably poor performance. Another reason is to capture the full diversity of data, especially data used for configuration. Complex CPQ applications can behave entirely differently depending on how they're configured. Subject matter experts need to be able to validate that your solutions function properly with the data they'll have in production.

If your org relies heavily on data for configuration, and you don't have some other mechanism to synchronize that configuration data, plan on refreshing your full and partial sandboxes relatively frequently. But bear in mind that if these orgs are kept in sync with external systems, you may have to update your org to point to those external systems after refreshing, and you may also need to resynchronize data with those systems or update records' external IDs to match the external systems. Challenges like this can make org refreshes onerous. Continuously seek methods to automate the post-refresh process, and balance the benefits of freshly synchronized data with the cost of performing refreshes.

Planning and Communicating Changes to Org Structure

Having determined your environment strategy, how you will use scratch orgs and how you will maintain and refresh your sandboxes, you need to ensure that anyone who works on those environments is aware of changes that might impact them.

You should establish a clear communication channel with all of the people impacted by changes to particular orgs. In large enterprises, you might identify subsets of stakeholders who work on particular environments. Establishing a shared calendar and repeatedly directing people's attention to it can help build awareness about scheduled changes.

Because refreshes and other org changes can interfere with people's work or even cause them to lose work, you should approach such changes carefully, emphasizing clear and regular communication in advance.

If you are unsure who is using a particular org and are planning to delete or refresh it, you can identify active users by using the Login History page in the Setup UI.

Working with Lightning Dev Pro Sandboxes

Salesforce has recently announced a new type of sandbox called Lightning Dev Pro sandboxes (LDPS). These have the same characteristics as normal Dev Pro sandboxes (1 GB of storage) but are designed to be created and destroyed using Salesforce CLI or API commands, just like scratch orgs. The enterprises that would benefit most from scratch orgs have also had the most difficulty in adopting them because of the challenges of untangling metadata dependencies into coherent packages. Thus most organizations remain focused on building with sandboxes.

Lightning Dev Pro sandboxes enable developers to create a pristine sandbox on demand using the CLI, modify it as needed, and capture their changes for deployment to target orgs. Soon it will also be possible to perform source synchronization for these orgs to make it simple to pull and push changes from these orgs. LDPS address the need for individual developers to have their own development environments that are up to date. The ease of creating and destroying these also helps ensure that developers can regularly restart their work in a clean environment that does not contain any "cruft" left over from previous development.

The Salesforce Upgrade Cycle

Salesforce performs a major version update three times a year. Their biggest update in any given year is their Winter release, which happens in the Autumn, but is named after the upcoming year (e.g., Winter '20 was released in the fall of 2019). This release is timed to coincide with Salesforce's mega-conference Dreamforce, the largest tech conference in the world, and many features are kept secret until they are revealed during Dreamforce keynote speeches, amid much fanfare.

Early in the new year, the Spring release is launched, followed by the Summer release sometime around June. In between these major version updates, minor patches are released to address bugs, performance, and stability issues. The schedule varies each year, but is posted in advance at `https://trust.salesforce.com`.

Each major release is accompanied by a new major version number in the API (e.g., Winter '20 is API v47.0, and Spring '20 is API v48.0). The API versions apply to REST and SOAP APIs, Apex classes, and documentation, among other things. Versioning their APIs allows Salesforce to change functionality between API versions while not breaking existing integrations and customizations.

These major releases are generally seamless and accompanied by a short 5-minute release outage. Most Salesforce users simply awaken to a new logo in the upper left of their org and notice few other changes aside from some new features.

Salesforce emphasizes that trust is their primary corporate value, and they go to great lengths to ensure that these major releases are indeed seamless and do not cause disruption for their customers. One aspect of their preparation for releases is an early access program in which the upcoming version of Salesforce is made available in some environments a month or so before it is released in production.

Getting Early Access to a Release

Prior to each seasonal release, Salesforce offers two methods for customers and partners to get early access to the upcoming release: prereleases and previews. Four to six weeks before a release, Salesforce makes it possible to sign up for prerelease Developer Edition orgs. These orgs provide the earliest access to upcoming functionality, although if issues are discovered, it's possible that features introduced in a prerelease will not be included in the eventual release; it's during this prerelease time that Salesforce runs their "Apex Hammer" regression tests to ensure they didn't break customers' functionality. Then, typically 2-3 weeks later, Salesforce opens up a sandbox preview window during which it's possible to create sandboxes and scratch orgs that use the impending version. Salesforce hosts webinars and publishes an updated blog post prior to each release which explains the details of the preview window for that release.

To get access to prerelease functionality, you can sign up to get a prerelease developer edition org by going to `www.salesforce.com/form/signup/prerelease-spring19/` (changing the URL to match the next impending version). That sign-up form is only available for a few weeks prior to each release, so you have to pounce if you want to get one. The good news is that prerelease org remains available to you just like an ordinary developer edition org, but is always upgraded to the next Salesforce edition before regular developer edition orgs are (I still use a prerelease org created years ago as my main org for learning and experimentation).

Preview sandbox instances require some understanding and planning from Salesforce admins, so you should ensure you are clear on the process. If you are managing your company's sandboxes, I strongly encourage you to read and understand the article called "Salesforce Sandbox Preview Instructions [for the … Release]" to

understand the process fully.[8] In brief, every sandbox is on either a preview instance of Salesforce or a nonpreview instance. The instance number of your sandbox determines whether it's a preview or nonpreview instance (e.g., CS21 is a preview instance, whereas CS22 is a nonpreview instance). Preview instances get updated about a month before the nonpreview instances.

As explained in the article mentioned earlier, the date on which you refresh your sandbox determines whether the sandbox is on a preview instance or a nonpreview instance, so the preview window will impact your sandbox refresh plans.

It's now possible to define whether scratch orgs created during the preview window are on the preview release or the previous release. By default, scratch orgs are created using the same release as the Dev Hub from which they're created. You can add the property "release": "Preview" to your scratch org definition file if you want the scratch org to be on the preview release, or "release": "Previous" if your Dev Hub has already been upgraded, but you want to create a scratch org that's on the previous release. These distinctions are only temporary; once the preview window closes, all orgs are upgraded to the latest release.

As mentioned, one thing that Salesforce does behind the scenes during this prerelease window is to run what's known as the Apex Hammer. Apex Hammer is a comprehensive execution of all Apex unit tests across all customer instances to ensure that they behave the same in the new version as they did in the previous version. One reason Salesforce implemented the requirement for customer Apex code to have 75% test coverage was to allow Salesforce themselves to perform regression testing on this code before seasonal releases. If Salesforce detects an inconsistency between your unit tests' output when run on different versions, that triggers an investigation and remediation. Again, this happens behind the scenes, but is a way of ensuring the reliability of the platform.

Deploying Between Environments That Are on Different Releases

Preview instances are very useful for getting early access to Salesforce capabilities and also to ensure that your customizations continue to work. You should take the preview window as an opportunity to revisit some or all of your custom functionality to ensure that it is still behaving as expected.

[8]The latest version of this article is always listed at `www.salesforce.com/blog/category/seasonal-release.html`

One challenge when working during the preview window is that it's possible to experience deployment errors if you attempt to deploy functionality from a preview instance (which may have new types of metadata) to a nonpreview instance. The Metadata API is versioned, so you can continue to retrieve metadata using the earlier API version number, even if you're now on the new version. However, sometimes new metadata changes can sneak in.

The most common issue I've seen is with new UserPermissions appearing in profiles. When you download a profile from a prerelease or preview instance, it may contain UserPermissions that don't exist in the current version. These will cause errors when deployed to an older version and so have to be manually removed from the profile for the deployment to succeed. As discussed in the section "Managing Org Differences" in Chapter 9: Deploying, it can be very helpful to have an automated method to strip out problematic tags arising from different org versions.

Sometimes a release will bring major changes to the metadata format. For example, during the transition between v37.0 and v38.0, the field type for all picklists was renamed `valueSet`, and the structure for managing picklists was changed in record types. Changes like this can require changes across the whole codebase to match the new format.

Prior to Salesforce DX, projects generally stored the API version for their metadata in the `src/package.xml` file for their project. That provided an explicit statement of the Metadata API version used for that metadata. Although projects can still specify a `sourceApiVersion` parameter for the project in their `sfdx-project.json` files, the default behavior is to determine the API version dynamically. It remains to be seen whether the Salesforce CLI handles this smoothly across every release, but it's at least now possible to quickly convert the entire codebase. Simply do an `sfdx force:source:push` of metadata in an older format, and then do an `sfdx force:source:pull` from a scratch org that's on a newer version of Salesforce, and you can expect to see the metadata format updated.

A Behind-the-Scenes Look at Orgs

One of the benefits of using Salesforce is that you don't have to know anything about its architecture; all you need to know is that it works and is sufficiently fast. But sometimes it can be helpful to have a bit of insight into the underlying architecture. In addition to impressing your coworkers, you can gain practical insights into topics like security, backups, and large data management.

I'm not an expert on this, and some of this architecture may be subject to change. Salesforce employs fleets of Site Reliability Engineers, sysadmins, database admins, and so on who help make all of this magic happen. They also share some of their insights and challenges publicly in Dreamforce sessions and on blogs like https://engineering.salesforce.com/. So look to those original sources for more information if you're curious. As mentioned in Chapter 2: Salesforce, the architecture described here applies only to Salesforce's core products like Sales Cloud and the Lightning Platform. The architecture for products like Heroku, Commerce Cloud, and MuleSoft will be substantially or completely different. As the product grows, there are an increasing number of additional aspects to this architecture, such as the Enterprise Messaging Platform (EMP) that are not explained here.

To make Salesforce core available, the company maintains nine of their own data centers, but also uses AWS to host their instances in some regions.[9] Salesforce announced a plan last year to partner with Google to use Google Cloud infrastructure to support Salesforce's international expansion.[10]

Within these data centers, Salesforce maintains over 100 "Pods" or instances known by names like NA42 and AP28. Each of these pods represents a self-contained set of resources to service a group of customers. Salesforce is a multitenant system which means that all customers use the same underlying database and application servers, but that data is segregated based on an Org ID to prevent the data or metadata for one customer from being accessible to or affecting that of another customer. This kind of multitenancy is at the heart of the cloud; from one point of view, "the cloud" is just mainframes with user partitioning.

Each of these Pods has a backup instance that can be used as a failover in the event that the main instance goes offline. There is even cross-data center data replication in the event that an entire data center goes offline.[11] Within each Pod there is a cluster of application servers and a cluster of database servers. When you first log in to Salesforce at https://login.salesforce.com, your request is routed to one of many authentication servers. Salesforce validates your credentials and then based on your username routes you to the appropriate Pod.

[9] https://help.salesforce.com/articleView?id=000257307&language=en_US&type=1
[10] https://cloud.google.com/salesforce/
[11] https://help.salesforce.com/articleView?id=000231130&type=1

Your interactions with a Salesforce instance are interactions with the application servers. These application servers provide standard functionality like record detail pages, as well as API access and any custom functionality such as custom apps, Lightning, Apex, or Visualforce pages you've created. To serve your data, these application servers reach out to the database servers to get access to the data for your org.

Importantly, all of your org's metadata including custom code is also stored in the database, in tables for standard and custom metadata. When you upload Apex code, that code is actually stored in the database to be executed as needed. Salesforce compiles page requests on demand, using the standard and custom metadata such as page layout definitions to structure the way that data is displayed in the output. There's a remarkable amount of custom processing that happens with each request to Salesforce, but they have managed to keep page load time relatively fast. This flexible structure is what allows admins to modify metadata and users to modify data simultaneously across thousands of orgs without impacting other tenants on the system.

The Salesforce database uses a generic structure to store all of the various custom fields and custom objects you create. They maintain one database table with records for standard objects. But any custom fields or custom objects you create are actually stored in a separate table. Records from all customers and all objects are stored together in those tables. One field is a globally unique ID (GUID) for that record, one field indicates the Org ID (the customer), and another field indicates which object the record is for (a lookup to the metadata table mentioned earlier). There are standard fields for CreatedBy, CreatedDate, SystemModStamp, LastModifiedBy, and LastModifiedDate, and the remaining fields hold the data for all the fields in that record. One reason why there are strict limits on how many custom fields you can have on an object in Salesforce is that Salesforce has a finite number of fields in the underlying database table. Each custom field you create consumes one of the available fields in that table, until eventually there are none left.

There are historical reasons for this structure: originally Salesforce didn't allow custom fields or objects, and so when those were added, they needed a new table to live in. Undoubtedly, there are good reasons why Salesforce has chosen to retain that structure. Presumably, this allows them to optimize the database performance for standard (known) objects while allowing for the table containing custom fields to favor flexibility over performance.

The net result of this is that Salesforce performs a number of table joins when querying data in your org. This leads to some of the behavior around query performance and opens the door for Salesforce to offer optimizations like skinny tables—a dedicated

table that speeds queries on very large data volumes by prejoining standard and custom fields. Salesforce maintains additional indexes for querying and searching this data.

The canonical description of this architecture can be found in this article by Steve Bobrowski, which is kept up to date over time: https://developer.salesforce.com/page/Multi_Tenant_Architecture.

Summary

It's important to understand the difference between Salesforce orgs and conventional servers, as well as the different types of Salesforce org. We've looked at how to secure and access orgs, as well as how to determine which environments are needed for your project.

Scratch orgs can be created "from scratch," but sandboxes are always cloned from an existing org. Sandbox cloning has become more versatile, allowing for new development patterns, and scratch org shape and snapshots promise to ease the scratch org creation process as well.

One key benefit of using Salesforce is that orgs are upgraded for you automatically, but it's important for teams to be aware of the timing of these upgrades and the impact on the development process.

Having understood Salesforce environments, the next chapter deals with the linear progression of changes through development, testing, and production: the delivery pipeline.

CHAPTER 7

The Delivery Pipeline

The delivery pipeline, or deployment pipeline, refers to the sequence of automated processes that manage the build, test, and release process for your project. The term was popularized by the book *Continuous Delivery*, which says:

> *A deployment pipeline is, in essence, an automated implementation of your application's build, deploy, test, and release process. … Every change that is made to an application's configuration, source code, environment, or data, triggers the creation of a new instance of the pipeline. … The aim of the deployment pipeline is threefold. First, it makes every part of the process of building, deploying, testing, and releasing software visible to everybody involved, aiding collaboration. Second, it improves feedback so that problems are identified, and so resolved, as early in the process as possible. Finally, it enables teams to deploy and release any version of their software to any environment at will through a fully automated process.*[1]

The function of the delivery pipeline is to facilitate the movement of features and fixes from development environments, through a testing process, and to end users. Thus the delivery pipeline connects the different types of Salesforce environments and is the path through which innovation flows.

Salesforce development projects always have a development lifecycle, but they do not necessarily go through a delivery pipeline. A delivery pipeline depends first of all on the presence of version control. On the foundation of version control, you must then have automated testing (to identify flaws and prevent them from being deployed) and automated deployments (to move functionality in a reliable and traceable way). This chapter discusses the basic foundations of version control and CI automation that pertain to any technology, while the later chapters in this section go into detail on testing, deployments, and releases on Salesforce.

[1]Jez Humble and David Farley, *Continuous Delivery* (Pearson Education, 2011), 55.

© Andrew Davis 2019
A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_7

Why You Need Version Control on Salesforce

At 6:45 pm PDT on Friday, May 16, 2019, Salesforce operations deployed a database script that impacted every org that had ever had Pardot installed.[2] This script inadvertently gave Modify All Data permissions to all users in those orgs, enabling every Salesforce user to access (and potentially modify) any data in that org. Such a change constitutes a security breach, and so out of an abundance of caution, Salesforce moved to lock users out of the affected orgs. But Salesforce didn't have an easy mechanism to prevent access to only specific orgs, so they took the "nuclear option" of disabling access to every org that shared a "Pod" (such as NA42) with an affected org. That left 60% of US Salesforce orgs inaccessible. Not only could these orgs not be accessed directly, but integrations that rely on them (including customer-facing web sites) were unable to contact Salesforce.

After 15 hours of downtime, access to these orgs was finally reenabled. But on the orgs which were directly impacted (those with Pardot installed), all non-admin users had their permissions entirely removed. Only admins were given access, along with instructions to restore the missing permissions.[3] Those instructions advised admins to restore permissions from a recently refreshed sandbox, or to restore them manually if they had no sandbox. But deploying profile permissions is notoriously challenging and without care and precision can easily lead to overwriting other metadata.

Salesforce is a remarkably stable and reliable system. But some of the admins affected by this outage will have struggled for days or weeks to restore their users' access. While no Salesforce user could have prevented the outage, the struggle to recover is entirely preventable.

My colleagues at Appirio manage the release process for a large medical device company. They had used version control and continuous delivery for the project from the very beginning, and their work formed an early foundation for Appirio DX. When access to their org was reestablished, they immediately began redeploying correct permissions to all their orgs. Even in the "fog of war," they were able to assess and repair the permission changes in just over 2 hours. Had they known exactly which permissions had been removed, they could have fixed things far more quickly. None of this would have been possible without a robust delivery pipeline being in place.

---

[2] https://status.salesforce.com/incidents/3815

[3] https://success.salesforce.com/issues_view?id=a1p3A000001SHDlQAO&title=some-user-profiles-and-permission-sets-were-modified-by-salesforce

Version control provides an extremely reliable backup for such metadata. And having a reliable delivery pipeline that you can use to restore configuration will protect you from extreme situations like this, as well as from a myriad of smaller problems.

Most of the problems that version control averts are not worthy of news headlines. But there is simply no substitute for having visibility into the history of changes on an org. And in large and small ways, that knowledge enables you to restore, diagnose, and experiment with complete confidence.

Version Control

Version control refers to a system for keeping track of different versions of a file and in its modern usage generally refers to software used to track versions of text files like source code. The most basic step that you and your teams can use to reduce risk while coding is to use version control on every single project without exception. Using version control is like keeping your money in the bank as opposed to under your bed. Banks protect your money, track all the inflows and outflows, and make it available anywhere to you and to those you authorize through ATMs and wire transfers. Similarly, version control means you never have an excuse for losing code, and all changes are tracked. That code is also accessible to you from any computer, and you can share it with those you authorize.

There are many types of version control technology, but in this text we'll be discussing Git almost exclusively. There are two reasons for this. First, I'm personally more familiar with Git than with any other type of version control. Second, Git has become overwhelmingly the most popular type of version control today.

It's important to note that Salesforce DX itself works entirely independently of the version control tool you choose to use. Many teams are successfully automating their Salesforce development lifecycle using TFS, Perforce, SVN, and other technologies. Most of the version control concepts shared here remain relevant regardless of the technology that you choose to use.

Git has its detractors and its disadvantages; it's more complicated to use than some competing tools. But it's undisputedly the dominant version control tool in the market, according to numerous developer surveys.[4] The Google Trends graph shown in Figure 7-1 shows that interest in Subversion (SVN) surpassed interest in Concurrent Version System (CVS) in 2005, while interest in Git surpassed SVN in 2011 and has continued to climb.

[4] https://rhodecode.com/insights/version-control-systems-2016

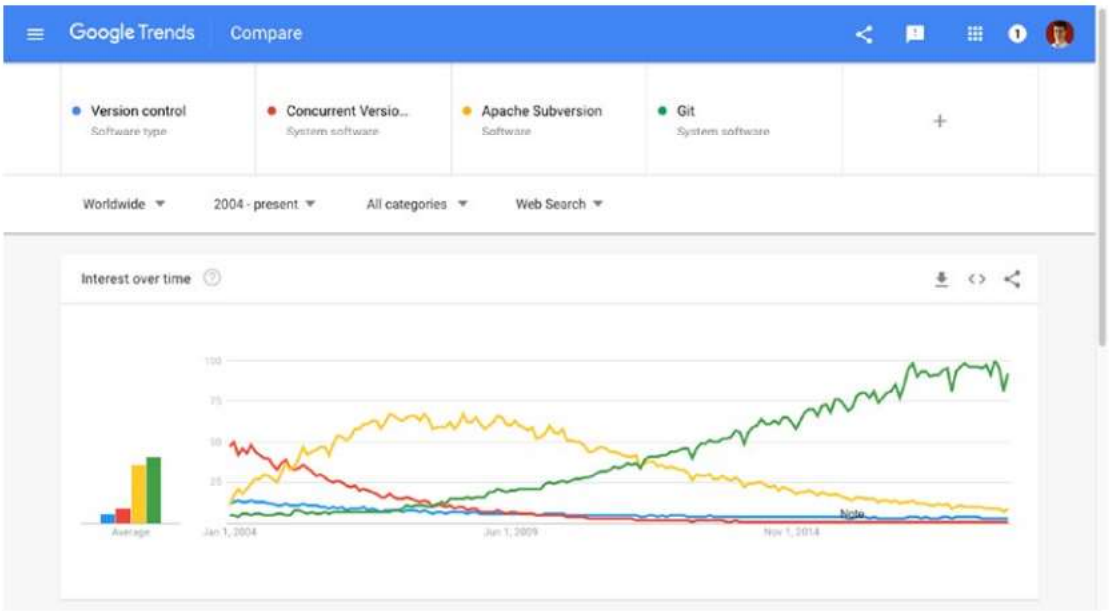

Figure 7-1. *Interest in version control in general, and CVS, SVN, and Git in particular, as measured by Google Trends in early 2019*

Git Basics

If you're not already familiar with Git, there are an amazing number of learning resources available to help you get started. In addition to endless articles, you can find interactive tutorials, videos, learning games, and live instructors. Even Salesforce Trailhead has a module on Git and GitHub.

Git itself is free and open source. It is an open technology that is interchangeable across all the different Git providers. People inevitably get confused, however, between Git and GitHub. GitHub is a commercial platform for hosting Git repositories that has become enormously popular over recent years. It faces competition from companies like Bitbucket and GitLab that also offer Git hosting. All these companies provide a secure, shared platform for teams to collaborate on Git repositories. They may offer a free tier, but they are commercial enterprises that charge for at least some of their hosting services.

Since Git is an open standard, it has also become possible for other version control systems and providers to provide Git-compatibility modes. For example, both Perforce and Microsoft Team Foundation Server (TFS) have their own proprietary version control technologies. But you can enable Git-compatibility mode on those code repositories

to allow developers to use TFS or Perforce as version control hosts while using Git commands and Git GUIs on their local machines.

Git is a distributed version control system, which means that every collaborator "clones" their own local copy of the code repository. Technically, every cloned instance of Git is identical to every other instance, and every cloned instance can directly share updates with any other copy of the repo. In practice, however, most teams establish a single central code repository (often hosted on GitHub or another Git hosting provider) and use that as the definitive copy of the repository. This creates a clear and simple client-server relationship between each developer's local repository and the central repository. This makes Git behave more like traditional version control systems like CVS and SVN, with the added benefit that you can easily work offline, since you have a complete copy of the repository on your own machine.

Having a shared, central repository is also the foundation for having a single, shared delivery pipeline. Although all of the automation done in a CI/CD system could also be done on a developer's local laptop, consolidating these processes on a shared service provides visibility and traceability, which are critical to reducing confusion about the state of your environments and builds.

If you haven't gone through a Git tutorial yet, here are the absolute minimum bits of knowledge you need to get started:

- Once you have cloned your project using `git clone <repository>`, you can create branches following the branching strategy of your project:

  ```
  $ git checkout -b <branch name>
  ```

- Once you have created a new branch, you are free to start making changes. To see the changes you have made, run

  ```
  $ git status
  ```

- Before you can commit the changes, you need to move them to a staging area. To stage the files you want to commit, run

  ```
  $ git add <file_name>
  ```

- To commit the staged changes, run

  ```
  $ git commit -m <commit message>
  ```

- Finally, to push the changes to the remote repository, run

  ```
  $ git push
  ```

- If you want to retrieve the latest changes present in the remote repository, run

  ```
  $ git pull
  ```

- This will merge the remote changes into your local copy. When you pull, your repository should have no uncommitted changes.
- If you and someone else made changes to the same lines in the same file, you'll get a merge conflict that you'll need to resolve manually. VS Code and other Git GUIs offer built-in support for merge conflict resolution, which can make it easier to understand and resolve the conflict.

Figure 7-2 summarizes these commands. If you understand this diagram, you have enough knowledge to get started with Git. But going through a thorough tutorial is well worth the time.

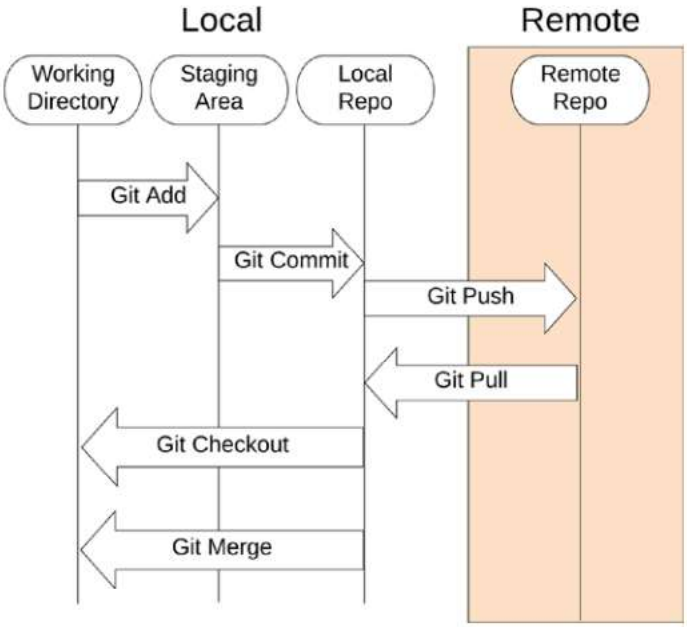

Figure 7-2. *Summary of the basic actions in Git*

Git Tools

There are many different ways of working with Git. The most common are summarized here for your convenience.

The classic way of using Git is to download the tool and run it from the command line as shown earlier. All of the other tools mentioned here actually use the Git command-line interface behind the scenes to work their magic. It's important to have a general understanding of how Git works on the command line, and even GUI users may find they need to perform commands on the command line occasionally.

Most Git tutorials explain using the command line. It's important to know how to run Git commands in that way, but I personally find it easier to use a graphical interface such as VS Code or SourceTree (`www.sourcetreeapp.com/`) for most daily operations. You can perform most Git commands using any of these tools, and you can mix and match these tools. For example, you could clone a repository using the command line, view the history of changes to that repository using SourceTree, make new commits using VS Code, and create a merge request using GitLab. All of these tools use the same underlying mechanism, so they stay in sync with one another naturally. I recommend that you get familiar with all of these tools because they excel at different tasks.

Git Settings

Git is a sophisticated tool, but the overwhelming majority of use cases are handled by a very small number of commands and settings. Earlier, I explained the basic commands. There are just a few settings you should be aware of. There are four common settings that almost every Git tool will walk you through setting initially: `user.name`, `user.email`, `core.autocrlf`, and `core.safecrlf`. These settings can differ per repository, but are typically just set once globally. The format for setting these is

```
git config --global user.name "Your Name"
```

The CRLF settings are in place to handle line ending differences between Windows and Mac/Unix systems. Auto CRLF will convert your line endings automatically, while Safe CRLF ensures that Git doesn't make any line ending changes that it can't reverse. The standard recommended settings for Mac/Unix are

```
git config --global core.autocrlf "input"
git config --global core.safecrlf "true"
```

And for Windows are

```
git config --global core.autocrlf "true"
git config --global core.safecrlf "warn"
```

When Salesforce teams are first migrating to version control, it's not uncommon to encounter line ending errors, especially with static resources which are uploaded from a user's local machine. There's a well-known utility called `dos2unix`[5] that you can use to batch convert the line endings in your project files. This is usually a one-time action when you first add files to the repository. After that Git takes care of the line ending conversions.

There is one Git setting that I've found tremendously helpful in dealing with the large XML files in Salesforce projects. Changing your Diff algorithm to "patience" ensures that Git takes extra care in comparing old and new versions of a file. XML contains lots of repetitive blocks, so this setting makes the actual changes more clear.

```
git config --global diff.algorithm "patience"
```

[5] http://dos2unix.sourceforge.net/

Git GUIs

Git GUIs like SourceTree, Tower, GitKraken, or GitHub Desktop excel at showing you a visual overview of the entire repository. You can quickly review changes across commits and branches and see changes to many files at one time. Release Managers and Tech Leads might find these tools particularly helpful since they need to review changes across many developers.

SourceTree is a free desktop application built by Atlassian that provides a UI for managing a local Git repository. It facilitates all of the Git features such as branching, committing, and merging. SourceTree makes it easy for new users to get started with Git, and it provides experienced Git users with a powerful way to manage even the most complex Git repositories.

Even if you're a command-line Ninja who was using computers before GUIs even existed, I'd strongly recommend you get familiar with one of these graphical tools. Especially if you're developing on a shared sandbox, you'll have to sort through metadata changes distributed across tens or hundreds of files, and you may need line-level precision when choosing what to commit and what to omit. Scratch org development makes the process far simpler, but graphical tools allow you to review changes and history with a speed and precision that are hard to match on the command line.

Git Embedded in the IDE

IDE plugins, such as VS Code's native Git support, excel at allowing you to make changes to files and then commit them quickly as you're working.

VS Code has built-in support for version control, which makes it an easy way to integrate the use of Git or other version control technologies into the development process. VS Code features a simple "sync" feature that pulls from the remote repo, merges, and then pushes your changes in just one click. It also has a great editor for resolving merge conflicts.

Git on the Command Line

Some people prefer to use Git on the command line, and it can be useful for everyone if you need to run less common Git commands. But be careful! Make sure you know what you're doing before just running a Git command that you found on the Internet. Some commands can have undesired consequences.

Git Host Web Interface

Working on the repo through the web browser interface of a Git host like GitLab can be useful for making quick changes, creating pull/merge requests (which are not a native Git capability), reviewing and commenting on other's code, or monitoring the progress of continuous integration jobs triggered from that Git host.

Git hosts provide a UI to navigate the Git repository as well as additional features such as merge requests for approving and merging changes. These hosts typically have many features to enrich collaboration, such as being able to add comments and have discussion around particular lines of code or merge requests.

The web interface can be a convenient way to solicit contributions from less technical members of the team. You can invite help from colleagues just by giving them access to GitHub, Bitbucket, or GitLab, without them having to download or learn Git. On the Appirio DX project, we passed documentation updates to an editor who used the web interface for her review.

Naming Conventions

The purpose of using a version control system is to make collaboration, experimentation, and debugging easier by providing a clear history of changes. Since this is a form of communication (across the team and across time), it's important that this communication be clear and done in a consistent way.

Every commit requires a commit message. Ideally, every commit represents a single, meaningful change to fix a bug or add a feature and is accompanied by a clear explanation of the change. In practice, unless a team is very disciplined about this, most commit histories are littered with unhelpful messages. I'm personally guilty of a lot of hasty messages like "fix the build." Similarly, a single commit might contain multiple unrelated changes, or conversely, related changes may be spread across multiple commits. While it's not always necessary for a commit history to be clean and clear, there are several approaches to commit messages and branch names that can help add value.

Commit Messages

Assuming that you're working in a team which is tracking their work in a ticketing system, each commit message should make specific reference to the ticket number that the commit is relevant to. This helps you track code changes back to the requirement

(user story, issue, or task) that prompted the change. This "requirements traceability" can also be important for complying with legal regulations. The ticketing system generally indicates who created a particular feature request and who approved it, while tying a commit message to the ticket indicates when that change was implemented in the codebase.

```
$ git commit -m "S-12345 I-67890 Added region to Account trigger"
```

If you are building CI automation around your commit messages (such as updating the external ticketing system based on a job status), be aware that commit messages can be truncated in the CI system. For this reason, it is helpful if the ticket number is placed at the beginning of the commit message. Any story/issue/task numbers used toward the end of a long commit message may be truncated and so not be available to the automation.

As various Git hosts have evolved their feature offerings and competed with each other, many of them have built integrated ticketing systems such as GitLab issues or GitHub issues. Similarly, Bitbucket provides native integration with Jira and Trello, also Atlassian products. These integrations allow for deeper integration between the ticketing systems and Git commits. For example, Atlassian allows you to view all related Bitbucket commits directly inside Jira. GitHub and GitLab allow you to close issues on their boards by referencing issues in commit messages such as "… fixes #113."

Feature Branch Naming

If you're using feature branches, the name of your feature branch is included by default in the commit message when you merge the branch into the master branch. This means that the names of your feature branches impact the commit history and can be used to make the history more meaningful.

Git GUIs such as SourceTree will recognize slashes in feature branch names and show them as a tree. Thus you can give your feature branch a detailed name that is prefixed with `feature/` to have all similarly named branches grouped together. To make it clear to everyone on the team what this branch pertains to, it's a good practice to include the ID of the work it relates to and a brief description. Teams with large numbers of developers each working on individual branches may find it helpful to include the name of the developer in the branch like `feature/sdeep-S-523567-Oppty-mgmt`.

- Using the `/` in `feature/` allows those branches to be grouped in SourceTree like a "folder."
- If your team has large numbers of branches assigned to individual developers, including your name (e.g., `sdeep`) causes your branches to be sorted together and makes it easier for you and your colleagues to identify your work in progress.
- Including the work ID (e.g., `S-523566`) will uniquely identify that work and allow you to automatically reference commits in that branch from your ticketing system.
- Including a description (e.g., `Oppty-mgmt`) helps humans (like you) identify what that work is about.

Some commercial tools like Appirio DX facilitate quickly adhering to such naming conventions.

Techniques such as Git Flow make use of a variety of branch types, which benefit from following a consistent naming convention such as hotfix/, release/, and so on. As discussed in the following, this is generally a less efficient workflow than trunk-based development.

Squash Commits

Another useful capability if you're developing in feature branches is the ability to squash commits. Squashing commits means to combine the changes from multiple commits into a single commit. This allows developers to make numerous small commits as they iterate through code changes while still creating a public history of commits that is succinct and meaningful. GitHub and GitLab both allow you to merge a branch into a single "squashed" commit, with a more meaningful message.

Semantic Release and Conventional Commits

There are several initiatives to enforce and make use of more rigorous commit message conventions. Probably the best known approach is called "Semantic Release," which uses a commit syntax called "Conventional Commits." There are tools such as Commitizen that can help you to enforce these conventions.

In Conventional Commits,[6] every commit message begins with a keyword indicating what type of change this is. The main keywords are `fix` indicating an issue fix, `feat` indicating a feature, or `BREAKING CHANGE` indicating a change that is not backward compatible. Several other keywords are widely accepted such as `chore`, `docs`, `style`, `refactor`, and `test`.

Following the **type** keyword, you can add an optional scope, indicating which part of the system you're modifying. The type and scope are followed by a colon and then the description of the change, so that the commit message would read like these examples:

```
feat(lang): added Polish language
docs: correkt speling of CHANGELOG
BREAKING CHANGE:extendskey in config file is now used for extending other
config files
```

Semantic Release builds on this convention to enforce the semantic versioning standard. Semantic versioning (semver) is the convention where version numbers are expressed as major.minor.patch numbers (e.g., 2.1.15). According to the semver proposal, if you are releasing a fix, you should increment the patch version number; if you are releasing a new feature, you should increment the minor version number; and you should only increment the major version number if you are implementing a breaking change. So, for example, Gulp 4.0.0 is not backward compatible with Gulp 3.9.1, but Gulp 3.9.1 **is** backward compatible all the way back to Gulp 3.0.0.

Semantic Release aims to enforce this numbering convention by updating version numbers solely based on commit messages. Semantic Release provides plugins that work with different technologies and version control hosts to enable version numbers to be incremented automatically based on commit messages.

To help your team enforce these naming conventions, tools like Commitizen provide interactive prompts as shown in Figure 7-3 which allow you to specify the type of change from a dropdown list before prompting you for the scope, a description, and whether the commit referenced any issues in the ticketing system.

[6] www.conventionalcommits.org/en/v1.0.0-beta.2/

```
PROBLEMS   TERMINAL   ...                    2: git

~/c/a/appiriodx-desktop-app> git cz                                  master+
cz-cli@3.0.5, cz-conventional-changelog@2.1.0


Line 1 will be cropped at 100 characters. All other lines will be wrapped after 100
 characters.

? Select the type of change that you're committing:
  feat:     A new feature
> fix:      A bug fix
  docs:     Documentation only changes
  style:    Changes that do not affect the meaning of the code (white-space, format
ting, missing semi-colons, etc)
  refactor: A code change that neither fixes a bug nor adds a feature
  perf:     A code change that improves performance
(Move up and down to reveal more choices)
```

Figure 7-3. *Commitizen provides a series of prompts to help you write consistent commit messages*

Preserving Git History When Converting to Salesforce DX

When you begin working in Salesforce DX, you will convert your project files from the Metadata API format into the Salesforce DX format. If you have an existing code repository and want to keep your Git history, you can follow this process.

The metadata conversion process copies your metadata into a folder such as `force-app/main/default`. Some types of metadata such as custom objects are then "decomposed" into smaller files.

To retain a history of these files in version control, you should delete your original `src/` files at the same time that you commit the new Salesforce DX files. In most cases, Git will recognize this (deletion of an old file and creation of a corresponding file) as a rename operation and your history will be preserved.

Stage these "rename" changes before making any other changes to the metadata. In some cases, Git will not correctly identify the origin and destination files. In those cases, you can stage and commit "renamed" files in smaller batches. For example, you can stage only the deletion of `src/classes/` and the creation of `force-app/main/default/classes/` and commit them together as a single commit. You can then proceed with committing other metadata types, one batch at a time.

Note that you cannot preserve history on the "decomposed" metadata. Classes and workflow rules are simply moved to a different folder or renamed slightly. But Object files are broken into many smaller files. The complete object file will appear as deleted in version control, and many smaller files will appear in its place. Nevertheless, even after they're deleted, you can go back to the earlier history for those files should the need arise.

After you commit these changes, you will almost certainly want to divide the source code into subdirectories or multiple repositories to enable modular/package-based development. Git recognizes file movements as a rename operation, but relies on the file contents to be relatively unchanged. So if you are simply moving metadata files into a separate folder, you should first save and commit any changes to those files and then move the files into the other folder and commit that change before proceeding with further changes.

If you decide you need to split your codebase into a separate repository, I recommend cloning or forking the existing repository rather than just creating a new repository. This will preserve the history of the files in both repositories. Clone the "parent" repository to create a "child" repository. Then delete from the parent repository any metadata files that should now belong only in the child repository. Finally remove from the child repository any metadata that belongs only in the parent. Commit those changes and you will have divided your codebase while still retaining the history.

Branching Strategy

One beautiful capability of version control systems is that they allow for the creation of branches or alternative versions of the codebase. These branches allow individuals and teams to share a common history, track changes in isolation from one another, and in most cases eventually merge those changes back together. Branches can be useful for experimentation, when one member of the team doesn't want their unproven changes to impact others on the team. And they can be useful for protecting one's codebase from the potential impact of other teams' changes.

Your branching strategy determines to what degree multiple variations of your code can diverge. Since version control drives automation such as CI/CD processes, branches can be used to drive variations in the automated processes that are run.

The use of version control is not controversial among professional programmers. The appropriate way to use (or not use) branches, however, is an almost religious debate.

Trunk, Branches, and Forks

Most code repositories have one main branch, which we can call the trunk. In Git, this is usually called `master` as shown in Figure 7-4. Other branches may branch off from this trunk, but they'll typically need to be merged back in eventually. This trunk is sometimes also referred to as the mainline.

In Git, even the trunk is just another branch. But in this context, we'll use branches to refer to versions of the code that separate off from trunk. In Git, a branch functions like a complete copy of the codebase, but one that can evolve separately from the trunk and from other branches. A short-lived branch is one that lasts for less than a day.
A long-running branch is one that lasts for more than a day. Because Git is a distributed version control system, every time someone edits code on their local copy of master, they are effectively working in a branch. However, as soon as they push their changes to the shared repository, it is merged into `master`, so (assuming that they push daily) they are effectively developing on the trunk. Only when a branch is given a name does it become formally separate from the trunk.

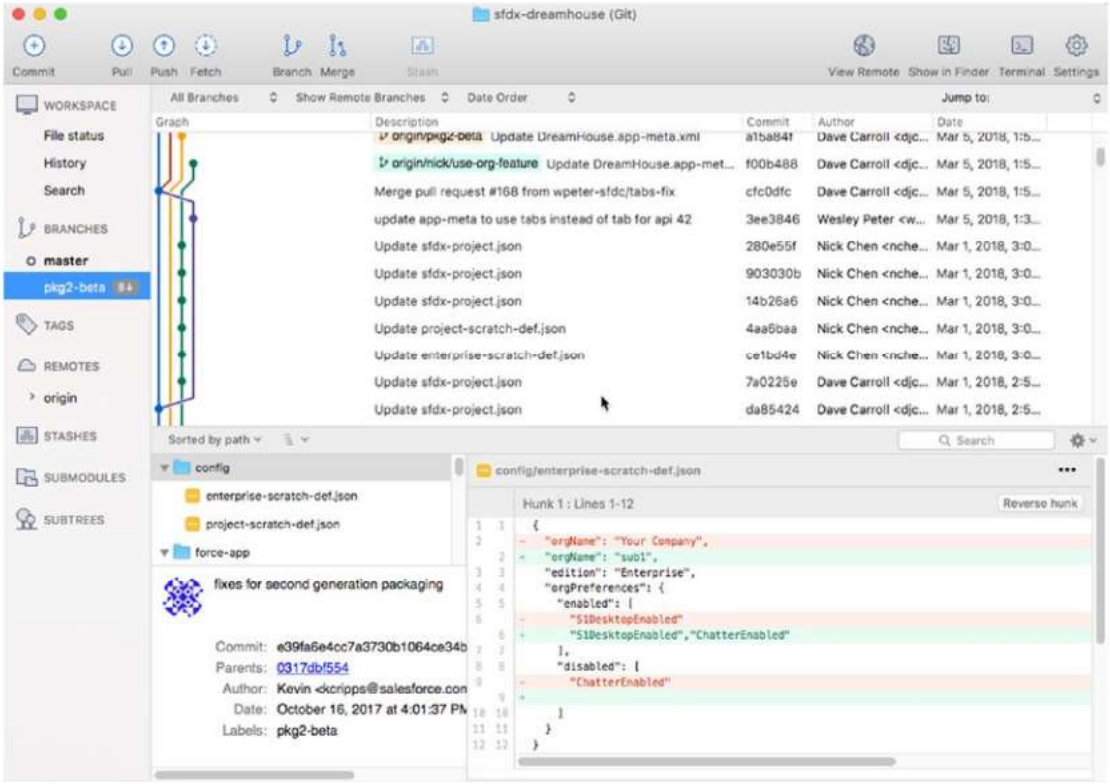

Figure 7-4. *Branching in the Dreamhouse app*

A Fork is a complete copy of a code repository that maintains a connection to the original repository. Whereas a branch begins as a copy of the trunk (or of another branch) but still lives in the same repository, a forked copy of a repository begins as a copy of an entire repository. This new repository is independent of the original repository and can evolve separately from it. But because they share a common history, changes from the forked repository can be merged back into the original repository or vice versa. In essence, a forked repository is like a super-branch: a branch of all the branches.

There are two reasons for forking repositories. One reason is when a team wants to take independent responsibility for their copy of a project. They might want to evolve it in a different direction (like the Jenkins CI server was forked from the original Hudson project), or they might want to use the project internally and customize it to their needs. The other reason for forking a repository is to securely allow for contributions to a project from unknown or untrusted contributors.

For example, Salesforce published their Dreamhouse repository on GitHub. They can allow their own developers direct editing rights on that project. But if a non-Salesforce employee such as myself wanted to contribute a change, they should not automatically trust that change. By forking the project into my own account as shown in Figure 7-5, I have full permission to edit this new copy. I can make changes and contribute a pull request back to the original project even though I have no permissions on the original repository. The team that owns that repository can then review the changes and accept or reject them.

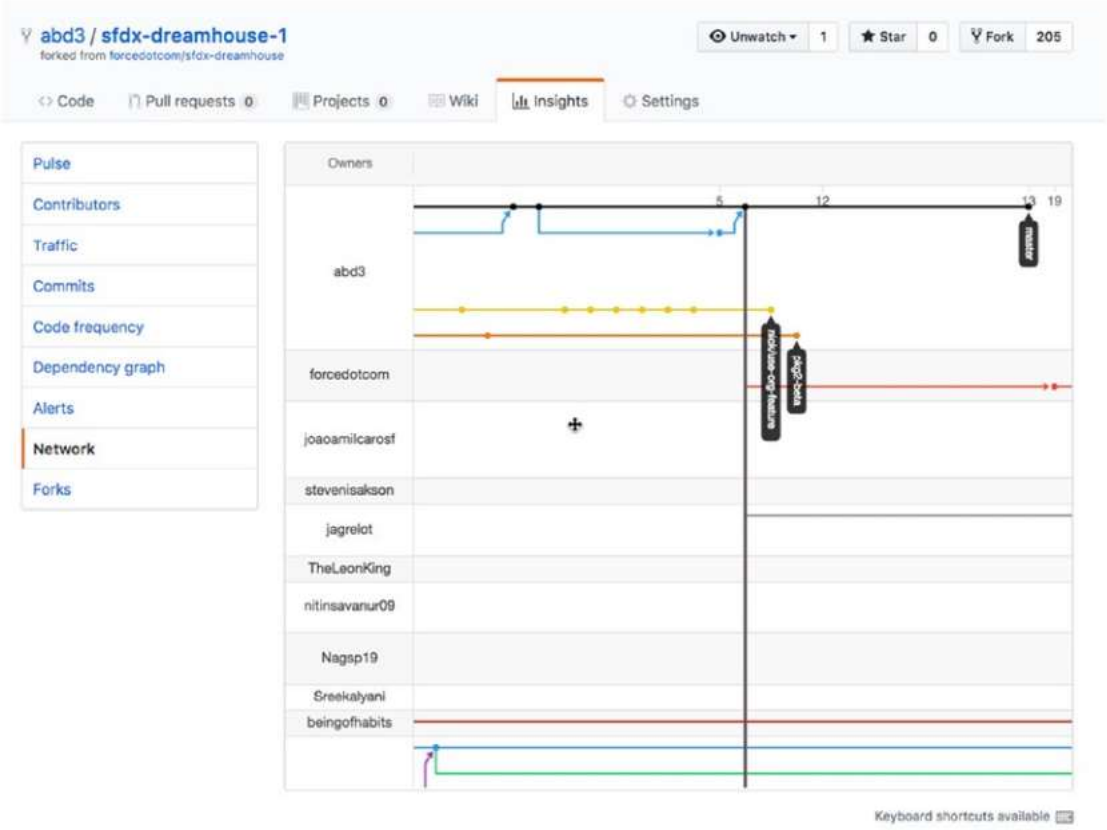

Figure 7-5. *A forked copy of Salesforce's Dreamhouse repository*

Well-Known Branching Strategies

There are many Git branching strategies that have been proposed and used. Table 7-1 lists the best-known strategies, along with my subjective view on them.

Table 7-1. *Brief summary of the best-known Git branching strategies*

Branching Strategy	My View
Centralized Workflow/ Trunk-based Development[7]	Simplest to understand, similar to SVN, a good place to start, the most efficient and proven workflow.
Git**Hub** Flow[8]	Uses feature branches and then merges them into trunk. This allows for systematic code reviews and is well suited to open source development with unknown contributors.
Feature Branch Workflow[9]	Very similar to GitHub Flow, but includes rebasing commits onto master.
Git**Lab** Flow[10]	Useful for versioned software. Generally more complex than needed.
Gitflow[11]	Sophisticated and popular, but tends to leave branches unmerged for too long, making it the antithesis of continuous integration.
Forking Workflow[12]	Useful when contributing to open source projects or highly complex, multi-repo programs. This compounds the risks of long-lived branches.

The Research on Branching Strategies

In the early days of enterprise software development, it was common for teams to work in isolation from one another for months or years at a time and then to have an integration phase involving weeks or months of risky and tedious merges of the codebase. The Extreme Programming[13] movement in the late 1990s popularized the practice of continuous integration, also known as trunk-based development,[14] in which teams performed the integration on an ongoing basis and worked together on a common mainline.

[7] www.atlassian.com/git/tutorials/comparing-workflows#centralized-workflow
[8] https://guides.github.com/introduction/flow/
[9] www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
[10] https://docs.gitlab.com/ee/workflow/gitlab_flow.html
[11] www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
[12] www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow
[13] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change* (2nd Edition). Addison-Wesley Professional.
[14] https://trunkbaseddevelopment.com/

The debate about whether to use feature branches or to develop together on `master` has been a long-running debate. Over several years, the DevOps Research and Assessment team analyzed the impact of branching strategy on a team's software delivery performance. The *2017 State of DevOps Report* shows their conclusions:

> *While our experience shows that developers in high-performing teams work in small batches and develop off of trunk or master, rather than long-lived feature branches, many practitioners in the industry routinely work in branches or forks. [Our study] results confirmed that the following development practices contribute to higher software delivery performance:*
>
> - *Merging code into trunk on a daily basis.*
> - *Having branches or forks with very short lifetimes (less than a day).*
> - *Having fewer than three active branches.*
>
> *We also found that teams without code lock periods had higher software delivery performance. (The ability to work without code lock periods is supported by the practices described above.)*
>
> *Additional analysis also showed:*
>
> - *High performers have the shortest integration times and branch lifetimes, with branch life and integration typically lasting hours.*
> - *Low performers have the longest integration times and branch lifetimes, with branch life and integration typically lasting days.*
> - *These differences are statistically significant.*
>
> *There's clear guidance to offer from these findings: Teams should avoid keeping branches alive more than a day. If it's taking you more than a day to merge and integrate branches, that's a warning sign, and you should take a look at your practices and your architecture.*[15]

Freedom, Control, and Ease

To understand why trunk-based development yields such benefits, and to understand what branching structure is most appropriate for your team, it's helpful to think about freedom, control, and ease.

[15] 2017 State of DevOps Report https://puppet.com/resources/whitepaper/2017-state-of-devops-report

In general, there is a tension between freedom and control—the more control, the less freedom; the more freedom, the less control. In an effort to overcome common problems in application development, it is typical for teams to oscillate between freedom and control as they define their processes. Where developers encounter too many obstacles deploying hotfixes to solve production problems, the team may give the freedom for certain changes to be expedited to production. Where individuals are seen to be making changes that break core functionality in production, additional controls may be put in place to prevent any changes directly in production.

What aspects of your version control system are conducive to freedom? Which are conducive to control? How should you best balance your options?

The fundamental idea of continuous delivery is that if you hope to have any control at all over your application and configuration, you need to be using version control, locking everybody out from changing metadata manually, and instead using an automated process for deploying those changes. That control comes at the expense of individuals being able to make ad hoc changes in any environment. But this level of control is a basic foundation that you should never compromise. Without version control and deployment automation, you will never have any real degree of control over your environments.

Once you control your Salesforce environments using version control and CI/CD, they function as a control panel for your development and release process. The guiding principle here is that you tune the freedom and control in your version control and CI system to determine the balance of freedom and control in your applications and environments.

What are the aspects of version control and CI that you can tune?

Your file and folder structure determines the content and structure of your application. If you divide your applications across multiple repositories, each of those repositories can control separate parts of the system. But forked copies of a repository and long-running branches within a repository allow for multiple alternative versions of that application. The major risk that branches and forked repositories present is that if they're all destined to control a single application, you will eventually have to merge them.

Merging can be risky. The more time elapses between a branch being created and merged, the more people forget the reasons why certain changes were made. And the more changes that are made on that branch, the greater the risk that the merge will cause conflicts or errors. These two factors, forgetfulness and the size of merges, are what make continuous integration (trunk-based development) a superior strategy in most cases.

The KISS principle applies (keep it simple and straightforward). Your files and folders should represent the structure of your application. Your commits represent changes to that application over time. CI jobs are used to manage stages of automation done on any given commit. That's three degrees of freedom already.

The advent of SaaS and auto-update technologies increasingly allow teams to only support one latest version of an application. If you have to support multiple versions of an application (imagine the patches to various versions of the Windows operating system), you will need to use multiple branches or repositories. But for modern applications like VS Code or the Salesforce CLI, for example, "only the latest version is supported." That's an enormous simplification and one that we should strive for with our own applications. One reason why Salesforce is able to innovate so quickly is that there are no "old versions" of Salesforce that their teams have to support.

Adding additional branches adds another degree of freedom … that you will eventually have to bring back under control. Better to do it straightaway, by not allowing the branch to last for more than a few hours. Adding additional forked repos adds yet another degree of freedom. Although I know teams that have taken this approach, and it is one that Atlassian's Salesforce development team used to recommend, in my view this can create an enormous amount of overhead and bring limited benefit. Forking repositories should generally only be used to allow untrusted developers to propose contributions to open source projects.

Merge requests (aka pull requests) are a way of adding control to a system by providing a formal approval process for branches being merged. Again, this is extremely useful for open source projects, but be very careful about adding merge requests as a requirement for your internal developers to submit a change to `master`. In practice, most tech leads do not review those changes carefully, so approving merge requests just delays the merging of feature branches and adds bureaucracy. Developers (and everyone else) definitely need feedback on their work, but real-time peer programming or informal code reviews generally yield higher performance without increasing the number of bugs in the system.

In addition to considering the balance between freedom and control, it's also important to improve the **ease** with which your team can deliver changes. The ease of managing your application's lifecycle depends on several factors:

1. Automating builds and deployments of all configuration eases the release process.
2. Automating tests eases the release process tremendously, in helping to identify bugs before they hit production.

3. And finally, tuning your approach to branching can help to ease the development process by not allowing any unnecessary variations to exist and addressing conflicts as they arise as opposed to allowing them to accumulate on long-running branches or forks.

Figure 7-6 summarizes the concepts in version control. Each aspect of this system represents a "freedom" or variation that can exist. Each aspect also provides an opportunity for control. You should tune your use of version control to maximize the ease of delivering changes while providing the appropriate balance of freedom and control. While branches and forks offer freedom to innovate without sacrificing control over the trunk, they bring significant inefficiency. Wherever possible, promote trunk-based development.

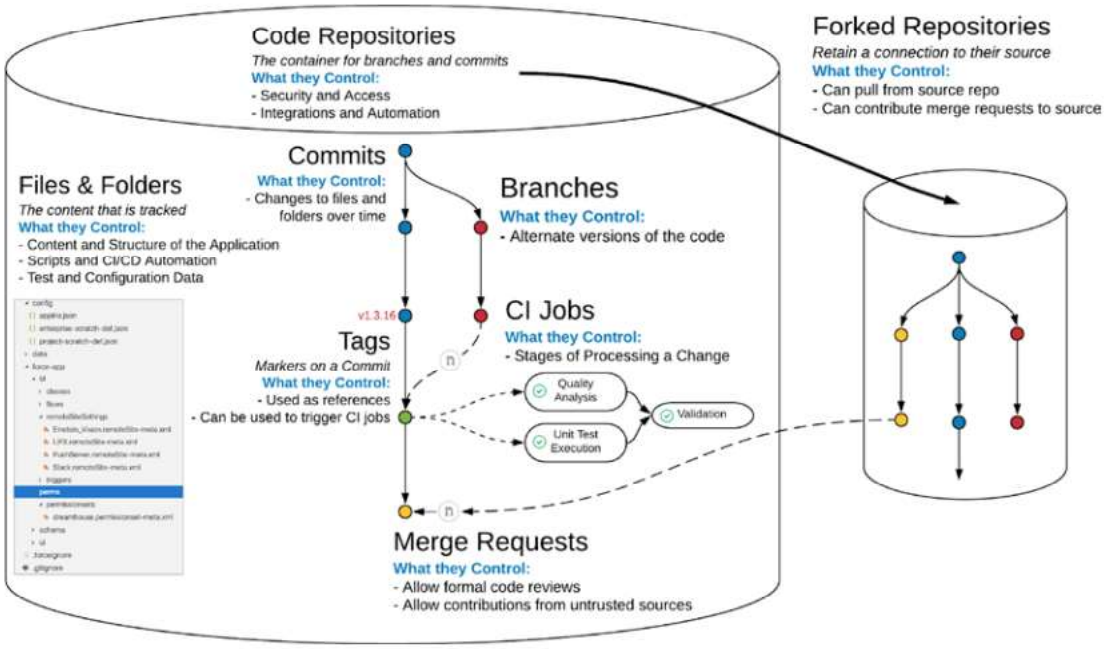

Figure 7-6. *An illustration of core version control concepts*

Branching for Package Publishing

Publishing Salesforce unlocked packages lends itself well to a simple branching strategy. Whereas for org-based development (see later), there are legitimate reasons why you may need to use long-running branches to manage differences across orgs, that's not necessary for managing packages.

Although unlocked packages and managed packages allow for a `branch` parameter to be specified, you'll need to take special care if you want to use that. If you are a managed package developer and need to support old versions of your package or different editions of Salesforce, there may be a need for a more complex process. But for enterprises developing unlocked packages, you should generally just use a single branch for your package's metadata, and allow its versions to reflect the linear history of improvements to the package.

In that sense, package versions correspond to the commits on the main trunk of your codebase or maybe to the subset of tagged commits. A simple, linear commit history corresponding to a simple, linear progression of package versions makes everything easy to understand.

The branch flag can be useful if you want to experiment on a separate branch in version control and actually publish package versions that won't accidentally be deployed to your main orgs. The branch parameter must be specified on the command line when publishing or installing a package version, and using this parameter doesn't require any changes to your underlying configuration files.

Salesforce DX allows you to develop multiple packages in one code repository by dividing them into subfolders. Your folder structure is thus the degree of freedom that allows you to represent multiple packages' metadata. While possible, this means that the security and build processes for that repository will apply equally to all packages unless you have built some sort of additional tooling. It is generally simpler to divide packages into multiple code repositories. That gives you the freedom to have different security and automation for the different packages; although you'll then need to manage their build processes separately and risk those processes getting out of sync.

Using one main trunk branch, you can still use feature branches and formal code reviews if you deem it necessary.

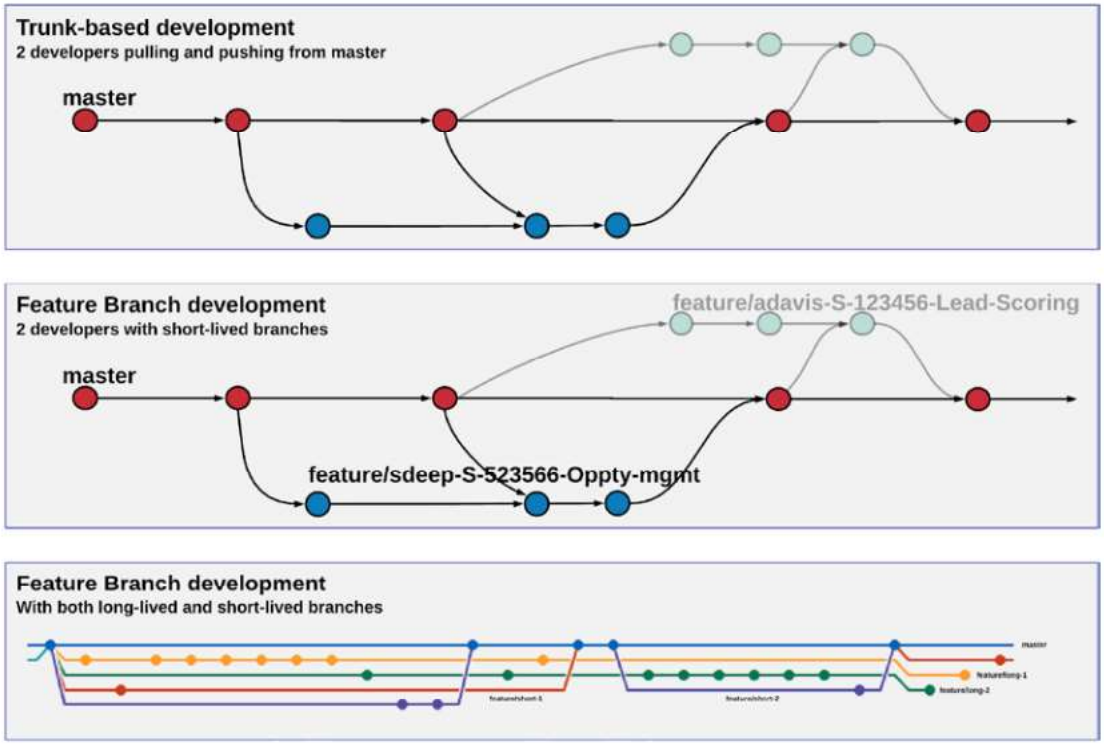

Figure 7-7. *Trunk-based development vs. feature branching*

As you can see in Figure 7-7, trunk-based development in Git is equivalent to using short-lived feature branches, with the difference that those "branches" are just developers' local copies of the `master` branch, so you can't run separate CI processes on those branches, although developers can run automated processes locally. You can also use Git hooks like pre-commit or post-commit hooks to enforce that certain processes are run with each local commit.

You can gradually implement tooling to support a refined CI/CD workflow around this branching structure. Common processes that you may want to run include

1. Static code analysis
2. Unit test execution
3. Manual jobs to allow the creation of review apps on a particular commit
4. Package publishing
5. Package installation in a target environment

The evolution of your tooling to support package publishing might look like this:

1. All packages in the repository are published whenever a commit is made to the `master` branch (generate a unique version number for packages by appending the CI job number as the build number).
2. To improve on this, only publish new package versions if the underlying metadata has changed.
3. To publish less frequently, you can have publishing be triggered only when a tag is added to the Git repo.
4. A sophisticated technique is to use semantic release to auto-increment version numbers and publish (or not) based on commit messages. This allows commit message syntax like "fix: handle null value exception" to auto-increment and publish a new patch version for the package. My colleague Bryan Leboff has published a Salesforce DX semantic-release plugin[16] to help with that.

Guidelines if You Choose to Use Feature Branches

There are several reasons why you might choose to use a feature branch workflow:

1. It allows you to use merge/pull requests to perform formal code reviews. Without the mechanism of merge requests, it is difficult to ensure that changes are reviewed and approved systematically.
2. It allows special CI jobs to be performed only on work in progress before it's merged into the `master` branch. Code in feature branches can be scanned for quality issues and validated to ensure it will deploy successfully before it's ever merged into the `master` branch.
3. It allows for in-progress features to be reviewed using review apps.

[16] https://github.com/leboff/semantic-release-sfdx

Although feature branches can be useful for these reasons, it is important to **avoid long-running feature branches**. Continuous integration is a well-established principle of software engineering which demands that code from different developers be merged frequently (at least daily if not within a few hours) so that conflicts can be detected and resolved quickly.

If you use feature branches, it is helpful to enable teams to create review apps based on a particular branch. This allows the branch to be previewed in a live environment for demo, review, or QA purposes. Reviewers can then give feedback before merging that code into the `master` branch. Changes related to one story are isolated from changes to any other work in progress, and the QA team can begin testing without waiting for a deployment to a QA sandbox.

If you use feature branches, it's helpful to delete them after they've been merged. This only deletes their name, not the history of changes. This reduces clutter, allowing only active branches to be listed in your Git tools.

See also the earlier guidelines on "Feature Branch Naming."

Before merging your work into the `master` branch, you should *merge the master branch into your feature branch* as shown in Figure 7-8. That forces each developer (who knows their code changes best) to resolve conflicts with the `master` branch (if any) rather than requiring the reviewer of any merge request to make judgments about how merge conflicts should be handled. When you're developing on the trunk, you have to pull and merge the latest remote version into your local repository before pushing your updates. Merging `master` into your feature branch is exactly the same concept.

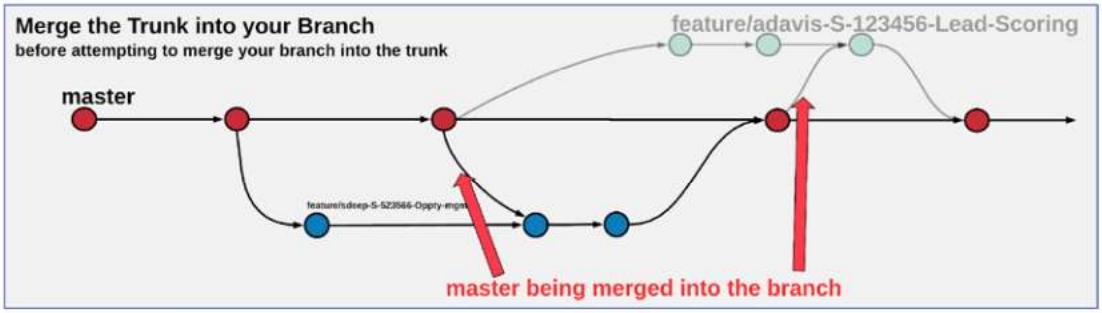

Figure 7-8. *Merging the master branch into a feature branch*

Branching for Org-Level Configuration

Just as it's important to use an automated process to track and build your packages, it's critical to also track and deploy your org-level configuration from version control. Prior to the development of Salesforce DX, managing configuration at the org level was the only way to implement continuous delivery for Salesforce. Therefore even for teams who are not yet ready to move to scratch orgs and unlocked packages, establishing a proper org-level delivery pipeline is extremely helpful.

Managing org-level configuration requires a different approach from package development. Orgs should generally contain identical metadata, but there are three important exceptions:

1. Some configuration (such as integration endpoints) needs to vary between the orgs.
2. Orgs are populated with metadata at different stages of development.
3. Some metadata in each org (such as most reports) does not need to be tracked.

How can you establish an automated process to keep orgs **identical yet different**?!

One effective approach is to have one folder to store the metadata that is common to all orgs and separate folders for each target org that can store metadata that is unique to that org. Deployments are made to each org by combining the common metadata with the metadata unique to that org. This requires some scripting to accomplish. Your desired end state is for most of your metadata to be contained in packages and for all orgs to be as similar as possible, so you should strictly limit the types of metadata that you put in the org-specific folders. NamedCredentials, RemoteSiteSettings, and org-specific Custom Metadata are great candidates to be put in those org-specific folders.

Secret values such as OAuth credentials should not be stored in the repository at all, but instead managed directly in the org or stored as environment variables in your CI system and injected into the metadata dynamically just prior to deployment.

Folders work well to handle common metadata and org-specific differences. But they are not a good solution for moving metadata through the stages of the development lifecycle. Because branches can be merged into one another, branches are the ideal solution for tracking metadata that should be sent to the testing environment now, but will soon be ready to move to the staging and then production environments. As shown

in Figure 7-9, you can use a testing branch to deploy metadata to the testing org; when you're ready, merge the testing branch into the staging branch and have that trigger a deployment to the staging org. Finally, do the same to production.

As a convention, you should use the `master` branch to refer to the metadata in production. The reason for this is that `master` is the default branch in your repository (the one you never delete); and while you may eventually delete or refresh sandboxes, you will never delete your production org.

From `master`, you can create branches that correspond to different sandboxes. You only need branches for sandboxes that are used to **test** features under development. Typically, organizations will have one, two, or three environments that are used to test features that have been released from development but still need to be verified. You only need sufficient branches to manage those variations.

Therefore not every sandbox should have its own branch. You can use a single branch to manage the configuration for multiple sandboxes, as long as you're happy to update those environments at the same time. For example, you can have an automated process that updates your training sandbox from the same branch used to update your staging sandbox. Whenever you deploy updates to the staging org, you can deploy updates to the training sandbox in parallel.

Although it's now possible to clone sandboxes, the default approach is to create and refresh sandboxes from the production org. When you create or refresh an org, this destroys any customizations made in that org. To reflect that, you can simply delete the corresponding branch. If the updated org has been cloned from production, you can then recreate the corresponding branch from `master`, since the sandbox is now in an identical state to `master`.

Bear in mind that the more branches you have, the more chance there is for those branches to get out of sync. Therefore, try to reduce the number of branches if at all possible. One small but important simplification can be made to the branch used to manage your final testing environment. The last environment used for testing should be your most production-like environment. Sometimes this is a full sandbox; sometimes it's called "stage" or "staging" to indicate that it's for configuration that is preparing to be sent to production. To say with confidence that you have tested your configuration before sending it to production, you can use the exact same branch (`master`) to deploy to this staging environment and to production. But whereas every commit made to this `master` branch will be deployed to this staging environment, you can use tags on this branch to selectively trigger deployments to production. These tags also provide

a quick reference to your production deployments. Using the same branch to manage both staging and production gives you ample confidence that you have tested your production changes in another environment. To be extra safe, you can use these tags to create a job in your CI system that you must trigger **manually** to actually perform the production deployment.

Finally, there are certain types of metadata that do not need to be tracked in version control. These are metadata like reports that you want to give your users freedom to create and edit directly in production. See "What's Safe to Change Directly in Production" in Chapter 12: Making It Better for a discussion of the types of metadata that do not need to be tracked.

An Illustration of This Branching Strategy

The branching strategy just described is one we have been using at Appirio since long before Salesforce DX appeared. This is not the only possible approach, but is one that is simple to implement with minimal automation. See Figure 7-9.

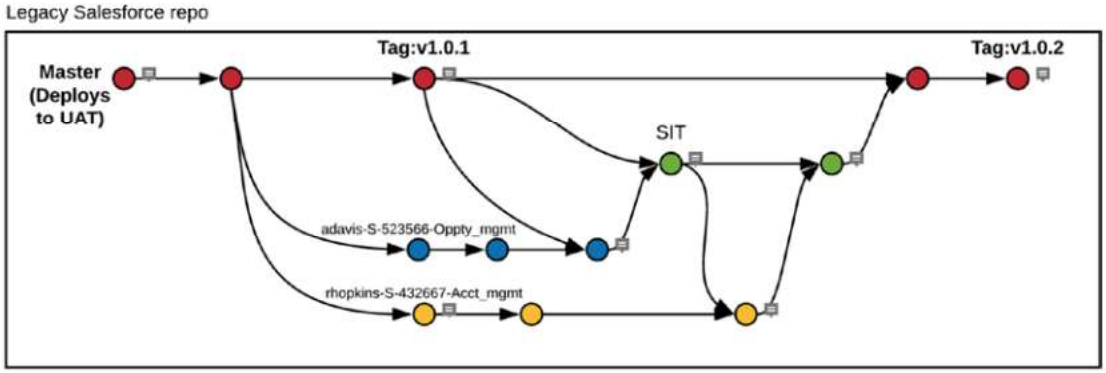

Figure 7-9. *Branching for org-level metadata*

We've found this approach generally works well and provides a balance between simplicity and power. To allow for formal **code reviews** and **branch-level automation**, many of our projects use feature branches while reducing complexity by limiting the number and lifespan of those branches.

The `master` branch is used to deploy code to the UAT org, and we use tags to deploy selected commits on the `master` branch to production. Every commit to the `master` branch triggers a deployment to UAT **followed by** a validation against the production environment. Additionally, if the commit to UAT contains a tag with a format like v1.0.3, this unlocks a manual option to deploy to production. In this way, we ensure that the

UAT and production environments contain the exact same codebase, and we never have to struggle with keeping those in sync.

Whenever a sprint starts, the Tech Lead or Release Manager should create an `SIT` (system integration testing) branch from the latest commit on `master`. When developers start their work on a story or issue, they create a feature branch from the latest commit to `SIT`. Once they commit their changes to their feature branch, a validation against the SIT environment is triggered. The developer can then go to the CI system to view the job status. If the validation job is successful, they then merge any recent changes to the `SIT` branch into their feature branch and create a merge request from their feature branch into the `SIT` branch. A dev lead can then view that merge request, see the code changes in the merge request, and approve or reject the merge request. If the merge request is accepted, the changes will be deployed to SIT and validated against UAT.

At the end of each sprint, the Tech Lead or Release Manager merges the `SIT` branch into master and **deletes** the `SIT` branch to allow this cycle to start again. Deleting and recreating this branch is very important since it reduces complexity in the repository. If you don't do this, branches will become increasingly hard to synchronize over time.

To sum up the process:

1. A developer makes a feature branch from `SIT`. As they work, they make commits to that feature branch. When they are ready to submit their work for review, they merge any recent changes from `SIT` into their feature branch and then push those changes to the repository.
2. The push to the feature branch triggers a validation against the actual SIT org.
3. If the validation to SIT is successful, they then make a merge request from the feature branch to the `SIT` branch. If there was a problem with the validation, the developer fixes that error and then pushes their changes again.
4. The merge request is reviewed by the tech lead and/or dev lead. They can see the lines edited/added on the merge request itself.
5. When the tech lead or dev lead approves the merge request, that code is merged into the `SIT` branch. That triggers a deployment of code to the SIT environment and validates it against UAT.

6. When a sprint ends, the `SIT` branch is merged into `master` and deleted.
7. A merge commit to the `master` branch triggers a pipeline which
   a. Deploys code to UAT
   b. Validates code against production
   c. Deploys the code to production if a tag with a particular format (e.g., "v1.0.3") is present

Branching Strategy with an Example

This example walks through the stages of this process:

1. The `master` branch of your project contains the codebase of the UAT environment. Let's say the two latest commits are as shown in Figure 7-10.

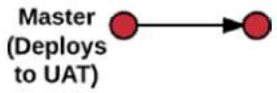

Figure 7-10. *Initial commits on the master branch*

2. To start any new development, you'll make a feature branch from the latest commit on the `master` or `SIT` branch. (In this case, we are starting with just a `master` branch). Let's spin up a feature branch as shown in Figure 7-11. Feature branches are nothing but branches which contain the prefix "feature" before the actual branch name. You'll name the branches as follows:

   `feature/[yourBranchName]`

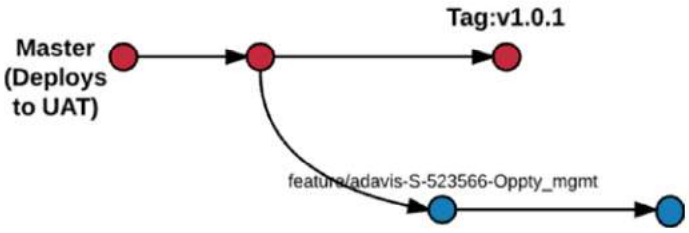

Figure 7-11. *Create feature branch from master*

3. Each commit you make to your feature branch validates the codebase changes against the SIT environment and displays any errors in the job status page in the CI system. You can access that by going to your CI tool's pipeline view. You'll see something like Figure 7-12 if the job passes.

Figure 7-12. *CI job status*

4. Moving on, let's say you're done making all the changes to your feature branches and your pipelines have succeeded. You're now ready to move your changes to the SIT environment.

5. The Dev Lead or Tech Lead should have merged the `SIT` branch into `master` and deleted it at the end of the previous sprint. They then create it again for the current sprint (this discipline helps ensure that the repo stays simple).

6. To simulate this process, create a branch called `SIT` from `master` as shown in Figure 7-13. Any commit to this branch will deploy the code straight to SIT.

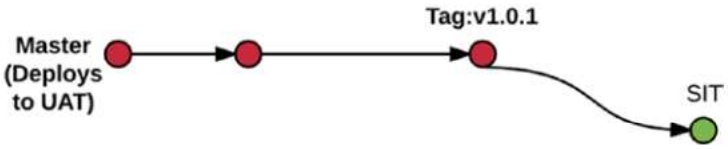

Figure 7-13. *Create an SIT branch from master*

7. Before merging your feature branch into SIT or any other branch it is important to first pull and merge the changes from the target branch into yours, fix any conflicts, and then push the combined changes. This ensures that each developer takes responsibility for bringing in the latest version of the shared code and ensures their changes are compatible with that.

8. Now, go ahead and merge `SIT` into your feature branch, resolve conflicts (if any), and make a merge request from your feature branch into `SIT`.

9. Go into your Git host's GUI to review that merge request. You can then approve the merge request to complete the merge into `SIT` as shown in Figure 7-14.

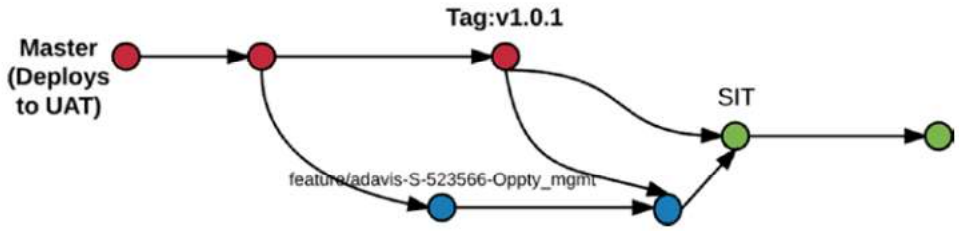

Figure 7-14. *Merging a feature branch into SIT*

10. Let's assume a sprint is over and you want to move changes to UAT. Make a merge request from the `SIT` branch to the `master` branch and approve it. This will deploy the code to UAT and validate it against production.

11. Create a tag on your commit to `master` called **vx.x.x** (where each x is a number between 0 and 9) as shown in Figure 7-15. This will initiate a manual deployment to production.

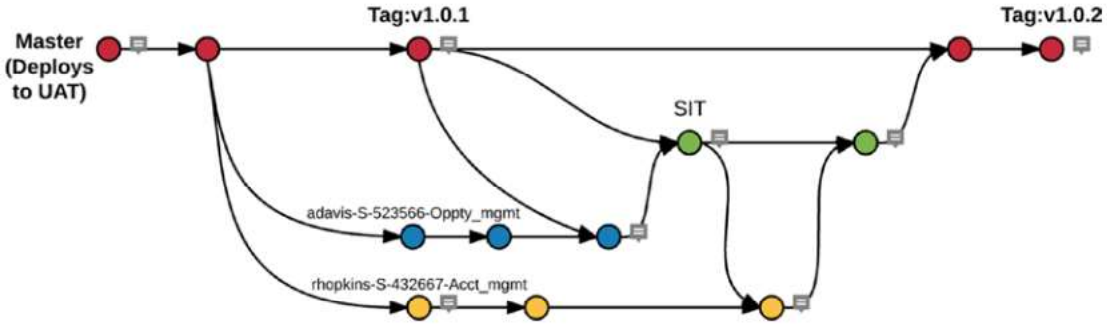

Figure 7-15. *The final branching structure for org-level metadata*

Deploying Individual Features

In pre-Salesforce DX CI processes, it was often difficult to elegantly move discrete changes from one environment to another. The org-level workflow just described (one of the only branching strategies that worked well) is focused on merging the entire content of test environment branches into the branch for the next stage of deployment. This works well as long as you're happy to move **all of the changes** from one environment to the next.

But inevitably some features are more important than others and need to be expedited before all features in that batch can be tested. It's unsafe to move large batches of untested changes into the next environment, just to expedite one of them. This is the fundamental problem with large batch sizes that lean software development strives to avoid.

The best solution to allow features to be deployed independently is to adopt unlocked packages. This allows package development teams to publish and install updates independently and thus allows certain features to be deployed more quickly than others.

Some commercial tools like Copado include powerful, automated branch management that allows feature-level deployment across any environment. If you're handling branching manually, there are two alternative approaches to managing the codebase at a feature level: cherry picking and feature branching. These approaches have been used by many teams prior to the availability of unlocked packages. Either approach may occasionally be useful to expedite certain features, but their disadvantages should also be understood.

Granular Feature Management Using Cherry Picking

One method to manage features independently is by making extensive use of cherry picking. Cherry picking is a native Git capability that allows you to apply a commit from one branch onto another branch without merging the entire branch. Deploying individual features by cherry picking will require significant discipline on the part of the development team. The basic concept is this:

1. Every commit should be explicitly tied to one work item, and the ticket number for that work item should be listed in the commit message.
2. Those commits can be made on a branch that is autodeployed to your first test environment.

3. When you're ready to deploy a feature to the next environment, you cherry pick the related commit(s) into the branch corresponding to the next environment.
4. Since commits depend on their predecessors, you should cherry pick commits in the order they were originally made and periodically merge the full branches to ensure that orgs remain consistent.

Challenges with this approach:

- Since this leads to environments being different, you are not actually testing an integrated production configuration before deploying it.
- It's easy to forget to cherry pick some commits. For this reason, you should still merge the full org branches as often as possible.
- Since commits are not really independent, features may not work properly if they are missing dependencies from other commits.
- This approach makes it easy for work to remain in progress for too long. This is a major warning sign in lean software development.

To enable certain features to be expedited to an environment, this approach can be used occasionally. But the risks in this approach make it inferior to building and deploying packages as a long-term strategy.

Granular Feature Management Using Branches

Another approach that can be used to manage features independently is to use feature branches to deliver features to each testing environment and to production. This is similar to the feature branch workflow described earlier, except that feature branches are merged not just into the first testing environment but also retained and used to merge into all subsequent testing environments and then into production. Although, in general, you will minimize the risk of merge conflicts if you first merge your destination branch into your source branch, if you follow this approach, it is important that you **do not merge the destination branch into the feature branch**.

Unlike cherry picking, merging branches brings the entire history of that branch's parents. Imagine, for example, that others on your team have merged five features into the `SIT` branch and that you have just completed work on a new feature branch.

As shown in Figure 7-16, if you merge the SIT branch into your feature branch prior to merging your feature branch into SIT, the other five features will then be included in your feature branch. If you later merge this feature into another testing org, you will be deploying both your own feature and those five other features.

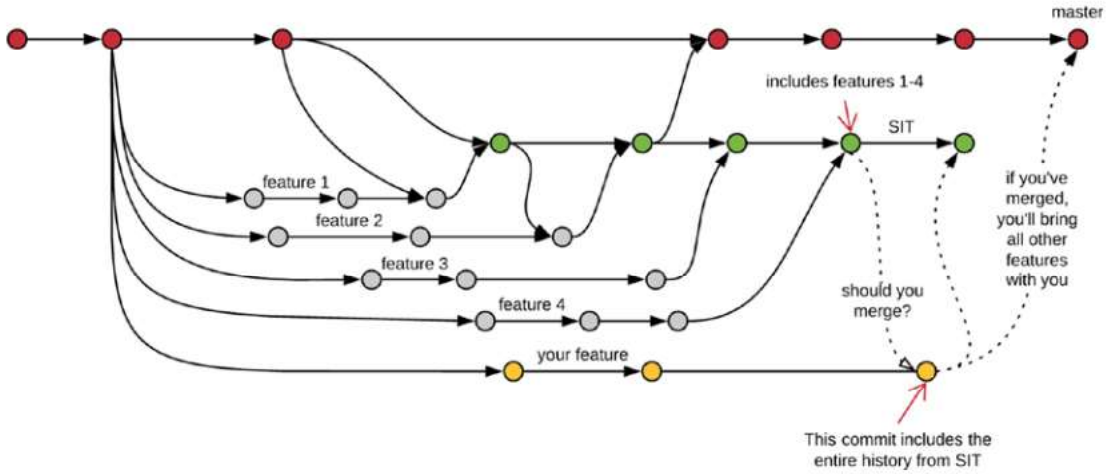

Figure 7-16. *Delivering feature-level granularity requires careful branch management*

Challenges with this approach:

- Since you can't safely merge your target branch into your feature branch, if you combine this approach with using merge requests for approval, this puts the burden of resolving merge conflicts onto the person approving the merge requests.
- This leads to a large number of long-running feature branches, which greatly increases the number of variations of your code. This tends to make every merge into a minor research project, as you assess whether that branch's code is up to date.
- It is harder for teams to understand and reason about the changes that will happen when branches are merged compared to cherry picking. Cherry picking moves just the changes in that commit. Branch merging brings all preceding changes.

- Such complicated merging can lead to subtle problems. If the person doing a merge chooses to ignore or remove some files from a merge, those files may automatically be excluded from future merges as well. This is due to the way Git calculates merges and can lead to very difficult-to-debug problems.

As mentioned previously, some commercial tools such as Copado can automate this entire process for you. If you're doing this manually, this approach may occasionally be useful for a very urgent feature (such as a hotfix deployment) but should be used very sparingly.

Forking Workflow for Large Programs

Trying to manage massive amounts of metadata changes at an org level can lead teams to adopt complex Git acrobatics. One of the most complex workflows is to use multiple forked repositories to manage the metadata for different teams. As mentioned earlier, forking a repository retains a connection to its source. This allows the team to perform merges between repositories. I've seen this pattern used when large numbers of teams each had their own sets of development and testing sandboxes, which were eventually merged into a shared staging environment and eventually into a single production environment. One repository was used to manage the staging and production environments, and separate repositories were used for each team's collection of sandboxes.

This approach allows different repositories to have different security restrictions (you can provide limited access to the repository governing staging and production) and allows each team to have a relatively simpler repository just to manage changes relevant to their team. For a time, this workflow was recommended by Atlassian in a site they had created to provide guidance on the Salesforce development workflow. The cynic in me wondered if their intention was to sell more code repositories although, in their defense, Atlassian Bitbucket does not charge per repository. This site has subsequently been taken down, but a number of Salesforce teams I'm aware of adopted this approach.

The inevitable consequence of this is that each repository drifts further and further apart. Be prepared to have multiple people working full time to handle upstream and downstream merges.

CI/CD and Automation

As mentioned earlier, continuous integration (CI) means that teams work on a common trunk in the codebase and run automation such as builds and tests with every commit to that trunk.

"CI" is often used to refer to tools such as Jenkins that perform automated actions on a schedule or based on code changes. In reality, tools like Jenkins are just tools for running and reporting on automated processes, and their use may be utterly unrelated to CI. These tools can be used to trigger any kind of job, either on a schedule, by a code commit, by an API call, or by some other mechanism. In spite of this, these tools themselves are frequently referred to as "continuous integration tools," and the jobs they run as "continuous integration jobs." The name has stuck and for our purposes is not inaccurate.

The reason for this word association, however, reveals an interesting history. In a situation where disparate teams work on different parts of the codebase and integrate near the end of a project, it is reasonable that the integration phase include extensive manual testing. The extreme programming movement promoted the practice of continuous integration as a more efficient alternative. Instead of having an extensive and error-prone integration phase, teams benefit from being able to integrate in small pieces on an ongoing basis. Conflicts are quicker and easier to resolve, overall code architecture can be more coherent, and teams can immediately benefit from each other's contributions.

But integrating code continuously in this way opens the door for regression failures to happen at any time. Any aspect of the system could potentially be broken at any time, and repeated manual regression testing is impractical. Continuous integration thus drives the need for automated testing. And for compiled languages, automated testing requires automated builds to be performed first.

Build automation is not a new concept, but in the early days, this was typically done on a schedule, such as a "nightly build." But a nightly build and test execution could mean a 23-hour delay in noticing that a developer's change had broken a test. To get that feedback as quickly as possible, teams moved to building and testing "continuously," every time the code changes. "Continuous integration" tools to automate these builds and tests thus became an essential component of the development workflow.

Once an automated system is in place for building and testing code, it's a small step to add deployments to that automated process. Thus the practice of continuous delivery grew naturally out of continuous integration.

To actually benefit from this automated test execution, you have to pay attention to the results. Simply putting in place automated builds and testing does not imply that the team is benefitting from this. For this reason, it's critical that teams develop the discipline of paying attention to the build status and fix failing CI jobs immediately as the top priority. A failing CI job, or "broken build," implies that changes have been made that cause builds, tests, or deployments to fail. Human short-term memory is notoriously unreliable, and so with each passing hour the team will find it harder to remember exactly what changes may have caused the failure. Of course version control gives a history that can be used to resolve these problems after the fact, but once a single error arises, all subsequent contributions will also fail. Thus a broken build is a blocking problem for the entire team. Practicing CI/CD implies not only that the team is merging their code continuously but also that they are ensuring the code can be built, tested, and deployed successfully at all times.

It's very common for people to believe they are practicing continuous integration just because their team has set up a CI server like Jenkins. Jez Humble is known for challenging this belief by asking three simple questions[17]:

1. Does your entire team merge their work into a common trunk at least daily?
2. Does every change to that codebase trigger a comprehensive set of tests to run?
3. If the build breaks, is it always fixed in less than 10 minutes?

If you cannot answer yes to those three questions, you are not actually practicing continuous integration.

Early detection and fixes are key to ensuring quality at the earliest possible stage. Developers should focus on setting up a simple CI/CD process as early as possible in the development lifecycle. Continuous delivery is the main technical capability at the heart of DevOps.

[17] https://continuousdelivery.com/foundations/continuous-integration/

Automating the Delivery Process

Using version control reduces much of the risk and tedium associated with the development process. But it is only when version control is tied to a CI system that its power to automate the delivery process is really unlocked. What are the elements in creating this automation?

CI Basics

Continuous integration is a process built around version control that allows code to be built, tested, and deployed every time it changes. To make this possible, everyone on the development team needs to be using the same version control system, and you need a CI tool that's configured appropriately for your project. A CI tool has three parts: the actual **CI engine** that orchestrates all the automated jobs, **configuration** that defines the specific jobs to run, and one or more **runners** that execute those jobs.

The role of the CI engine is to allow teams to create projects, give individuals access to those projects, define and configure one or more automated jobs for that project, trigger those jobs, and then monitor the jobs' status. The CI engine is generally the tool that you log into to see the configuration of a CI job or to check the status of that job.

CI job configuration determines what code is used as the basis for the job (e.g., which repository and which branch in that repository), when a job is triggered, what processes are run as part of that job, and what kinds of notification, reports, or artifacts are created as a result. Multiple jobs are often grouped into pipelines, in which the pipeline itself is triggered by some event, and it in turn triggers each job in the appropriate order.

CI runners are where the action happens. A CI runner is an isolated environment in which the CI job runs. The runner might simply be a dedicated folder on the main CI server, it might be a separate server or virtual machine, or it might be a container such as a Docker container. CI runners are typically separated from the main CI engine for three reasons:

1. Runners can be scaled out "horizontally," with one CI engine controlling scores of runners. Even if CI jobs are long running and resource intensive, they will never slow down the CI engine.

2. Runners can have entirely different hardware, OS, and software. iOS or Mac OS software builds usually have to be run on Mac hardware. For example, while most of the CI runners used at Appirio host Docker containers, we have some that run in Mac VMs, on Mac hardware.

3. CI runners provide security isolation, so that one team cannot view content on other runners or hack their way into the main CI engine.

In addition to creating logs, artifacts, deployments, notifications, and any other results, running any CI job always returns a pass/fail status. If jobs are arranged in a pipeline, subsequent jobs will not run if the preceding jobs have failed. This allows teams to, for example, not deploy code if the unit tests did not pass.

Pipeline Configurations

Most CI systems allow jobs to be organized into pipelines which define sequences of jobs. A pipeline is a group of jobs that get executed in stages or batches. All of the jobs in a particular stage are executed in parallel (if there are enough concurrent CI runners), and if they all succeed, the pipeline moves on to the next stage. If one of the jobs fails, the next stage is not executed (unless you've stated that job can "allow failure"). Pipeline configuration thus allows jobs to be organized in series, in parallel, or in some combination of the two. Specifying stages allows us to create flexible, multistage pipelines.

As mentioned earlier, you can specify under what circumstances you want particular pipelines to run. For example, you might want certain tests to run only when a commit is made to the `master` branch, and you might want commits to different branches to deploy to different environments. Making a commit on a particular branch can thus trigger a pipeline appropriate to that branch, which in turn triggers different jobs in a particular sequence.

CI pipelines are thus the mechanism used to manage a continuous delivery pipeline. Each commit triggers a pipeline, which triggers jobs for builds, tests, and deployments.

A pipeline is typically triggered when a commit is made to the repo, but can also be triggered manually, based on tags, based on a schedule, or using the CI engine's API, if it has one.

Multiproject Pipelines

A multiproject pipeline is a more sophisticated version of a pipeline, in which one pipeline may trigger other pipelines on other projects to run. This can be useful in many cases; one example is triggering the rebuild of one project whenever its dependencies are rebuilt. The dependency itself would trigger the parent project to be rebuilt in that case.

GoCD is an open source CI tool that was custom-built by ThoughtWorks, largely to handle the challenge of multiproject pipelines, since not all CI tools have this capability. GoCD itself is built using a multiproject pipeline (in GoCD!) that requires several hours to build and test numerous subprojects before compiling the final tool.

Seeing CI Results in Merge Requests

Merge requests (aka Pull Requests) are requests to pull and merge one branch into another. They create a formal opportunity to review code changes, and can also be used to review CI job results. For example, the CI process can perform automated validations on a feature branch; then a merge request can be created for validated feature branches to allow someone else to review and approve that branch before merging it into the `master` branch. Version control tools like GitHub, GitLab, and Bitbucket Pipelines can all show CI job status alongside the list of code changes in merge requests.

Environments and Deployments

Some CI systems use the concept of "Environments" to represent systems such as Salesforce Orgs that are affected by particular CI jobs. Where available, it's helpful to use these, since it gives a way to cross-reference which CI jobs (such as deployments) have run against particular environments.

CI Servers and Infrastructure

What are the different kinds of CI systems, and how should you choose between them?

Generic CI Systems vs. Salesforce-Specific CI Systems

Because of the historic challenges involved in managing the Salesforce development lifecycle, numerous Salesforce-specific release management tools such as Copado have been created. Increasingly, these tools are supporting and promoting the concepts of version control and continuous integration. These tools are dealt with in more detail in the section on "Commercial Salesforce Tools" in Chapter 9: Deploying.

In most languages and for most applications, the tools used to manage the development lifecycle are generic tools such as Jenkins. Jenkins can be used to automate the development lifecycle for any application in any language. When we refer to a "CI Server," we're generally referring to these generic tools, although the concepts usually translate to Salesforce-specific tools as well.

Although Salesforce-specific tools natively support Salesforce deployment and testing, they are often not as flexible as the generic tools. Using Salesforce-specific tools frees your team from the complex challenge of having to build and manage their own scripts, at the cost of some control and visibility over the complete process, as well as quite a few dollars.

The movement to Salesforce DX increasingly allows generic CI tools to support every aspect of the Salesforce development lifecycle. If your team has the ability to write or gather their own deployment scripts, they can create their own comprehensive process. You can also find some well-developed open source tools like CumulusCI to help.

Although generic CI tools are a viable option, as the person who oversaw the engineering team who built Appirio DX over the course of 2 years; I promise you that there are some unusual challenges involved in building Salesforce release tooling. Unless you have truly unique needs and a skilled developer tooling team, the total cost of buying a prebuilt solution will generally be far lower than the cost of building everything yourself.

Choosing a CI Server

As mentioned before, a CI tool has three parts: the actual **CI engine** that orchestrates all the automated jobs, **configuration** that defines the specific jobs to run, and one or more **runners** that execute those jobs.

Different CI tools handle those three parts differently, but this structure is more or less universal, and CI tools are more or less interchangeable. Most CI tools can handle most CI jobs and scenarios, so you can choose the CI tool that is most effective for your team. Both the *State of DevOps Report*[18] and the book *Accelerate*[19] present research on the importance of teams having autonomy to choose their own tools. Enforcing a single corporate standard for version control and CI can help ensure a company maintains expertise in those tools and can reduce overhead such as server or subscription costs. But it's important for teams to be able to deviate and implement their own tools if the benefits they bring outweigh the long-term costs of maintaining them.

Requiring all teams across a company to use a common CI server commonly leads to a bottleneck where the IT infrastructure team needs to be involved in every change

[18] https://devops-research.com/assets/state-of-devops-2017.pdf

[19] Nicole Forsgren, Jez Humble, and Gene Kim. 2018. *Accelerate: The Science of Lean Software and Devops Building and Scaling High Performing Technology Organizations* (1st ed.). IT Revolution Press.

to the CI server configuration. That limits the adoption of CI tools and a team's ability to experiment.

Some CI tools have an architecture that allows for teams to have full autonomy even if there is only a single instance of the CI tool. When selecting a CI tool, look for one that has these three characteristics:

1. The team has autonomy to control access to CI configuration such as job status and logs.
2. You can store CI job configuration as a code file inside your codebase itself.
3. You can run each CI job in an environment that the team controls, typically a Docker container.

It should be clear why the team needs the freedom to access their own job status and logs: DevOps implies that it's the team's responsibility to monitor the status of their own jobs and to use logs to debug failing jobs. As John Vincent famously said, "DevOps means giving a s**t about your job enough to not pass the buck."[20]

The benefits of storing CI configuration as code are discussed in the next section; but why is it so beneficial to use Docker containers to run CI jobs?

Why Use Docker Containers to Execute CI Jobs?

One capability that enables a flexible CI system is the ability for teams to control the environment used to run the CI jobs. CI jobs are simply processes that execute and report back a pass/fail result. To run, these processes need an execution environment that has all of the necessary software installed. In the early days of CI systems, that meant configuring a server and manually installing software in it. As teams' needs evolved, a server admin installed or upgraded supporting tools as needed. Jenkins popularized a plugin model that allowed teams to install common build tools like Ant and Maven from a plugin library. Jenkins still has the richest collection of CI plugins available. But plugins have to be installed and configured in the Jenkins instance itself. In large organizations, the project team may not have permission to install plugins themselves and may have to submit a ticket to IT and go through an approval process.

[20] http://blog.lusis.org/blog/2013/06/04/devops-the-title-match/

Recent CI systems have taken the extremely flexible approach of allowing CI pipelines and jobs to be run in Docker containers. Docker is a tool that enables the fast creation of lightweight execution environments ("containers") from images that are defined using simple configuration files ("Dockerfiles"). Dockerfiles always define a Docker image as a starting point. This allows a team to define a custom image that is based on an existing Docker image, removing the need to rediscover all of the software dependencies they might need. Docker images can be stored in a custom Docker repository, but an enormous number of predefined images are available on `https://hub.docker.com`. Most of these also link to the Dockerfile used to create them, making it trivial to recreate these images if you want complete control over how they are defined.

For example, the sample Dockerfile shown in Listing 7-1 is based on the official Docker image for Node.js version 8. On that basis, we use the Node package manager called Yarn to install Salesforce DX and then set a few environment variables. This Dockerfile is then built into a Docker image, which can be used to reproduce any number of execution environments, each of which has Salesforce DX installed and ready to execute.

Listing 7-1. myCompany/salesforceDXimage—a sample Dockerfile for Salesforce DX

```
FROM node:8

#Installing Salesforce DX CLI
RUN yarn global add sfdx-cli
RUN sfdx --version

#SFDX environment
ENV SFDX_AUTOUPDATE_DISABLE true
ENV SFDX_USE_GENERIC_UNIX_KEYCHAIN true
ENV SFDX_DOMAIN_RETRY 300
```

Docker images are almost always based on a Linux variation like Ubuntu. Windows images have recently become supported by Docker, but the OS version in the image must match the OS version of the host. This means that you can now create a Docker image that runs Windows Server 2016, but it can only run on a Windows Server 2016 host. Mac images are not currently supported.

The ability to run CI jobs in Docker containers is extraordinarily powerful. It means that there is no need to load the CI server up with endless manually installed pieces of

software or plugins. A CI configuration file can simply specify the name of the Docker image that it wants as its execution environment and what commands it wants to execute in that environment. When the job runs, the CI runner spins up a Docker container *exclusively for that job*, executes those commands, reports a pass/fail value, and then destroys that Docker container.

Any piece of software that you want can be available in that Docker container, and importantly you are guaranteed that the server environment is identical and clean every time the job runs. With Docker containers, there is zero possibility that one job might create a side effect that would change the behavior of subsequent jobs. Such side effects are extremely hard to debug, making Docker containers a simple and low-stress execution environment.

CI systems like Bitbucket Pipelines, GitLab CI, and CircleCI do an excellent job of caching Docker images. This means that although the first time a Docker image is used to run a CI job it might take a minute to download, subsequent containers can be created in (milli)seconds. Container creation is so fast that the uninitiated would never dream that each CI job begins with the creation of a unique new computing environment!

The (slightly contrived) GitLab CI configuration file in Listing 7-2 shows the power and simplicity of using Docker images. At the start of the file, we specify that the Docker image called `salesforceDXimage` (defined in Listing 7-1) should be used as the default image for running each CI job. Two pipeline stages are defined, each of which has one job. The first job, `create_test_org`, uses the default image and executes Salesforce DX commands to create a new scratch org and push source to it. The second job shows how you can override the default Docker image for a specific job. In this case, we run a Ruby script in a container based on the official Ruby image.

Thus this one CI configuration file allows us to make use of two entirely different execution environments. The `salesforceDXimage` defined in Listing 7-1 does not have ruby installed, and the `ruby` image does not have Salesforce DX or Node.js installed. The environments are entirely independent, yet we can make use of both and define the sequence and conditions under which they'll be used. If the first job in this pipeline fails, the entire pipeline fails and `run_tests` will not execute.

Listing 7-2. Sample .gitlab-ci.yml file for running tests

```
image: 'myCompany/salesforceDXimage:latest'

stages:
  - build
  - test

create_test_org:
  stage: build
  script:
    - sfdx force:org:create -a testOrg --setdefaultusername --wait 10
    - sfdx force:source:push
  only:
    - master

run_tests:
  stage: test
  image: ruby:latest
  script:
    - ruby -I test test/path/to/the_test.rb
  only:
    - master
```

Defining servers using Docker images has become the default approach for new IT projects and is the most flexible approach you can take for defining your CI processes as well. If you're using a generic CI tool, choose one that allows you to define and run jobs in Docker images.

Although Appirio DX is a commercial tool, it provides a Docker image `appirio/dx` that contains the Salesforce CLI and many other helpful tools. You can freely use or build off of that image as a starting point for your CI process.

Example: Using GitLab

At Appirio, we chose to use GitLab because of their fast pace of innovation and because a feature-rich CI system is built into the tool. Other version control platforms increasingly bundle CI tools, and CI tools increasingly support the capabilities mentioned earlier.

GitLab is available as a SaaS hosted service on `https://gitlab.com` or as a self-hosted service. Both the SaaS and self-hosted instances have free tiers, as well as paid tiers such as Bronze, Silver, and Gold.

GitLab itself is a version control tool. GitLab CI is the engine that orchestrates all the automation, and it's tightly integrated with the rest of GitLab. The configuration for GitLab CI is all contained in a simple text file called `.gitlab-ci.yml` that lives alongside your project's code in the repository; it's simple enough to be easy to read, but powerful enough to describe almost any CI/CD configuration. To set up CI for your project, you don't need to log in and configure anything; just drop a `.gitlab-ci.yml` file into your project repo and the whole CI system comes to life!

The GitLab **runner** typically hosts Docker containers—super lightweight Linux shells that allow us to run almost any kind of software and code that we want. You can create runners that do not host Docker containers as well. Most runners can be shared across many projects, but you may sometimes need to create special GitLab CI runners for specialized applications like building OS X apps or other Docker images.

Because GitLab CI integrates with GitLab, everyone who makes changes to the code repository is implicitly using GitLab CI. For example, just by creating and pushing a feature branch, a developer triggers the execution of any CI jobs that run on feature branches. Using CI is thus very easy, you just use version control and you're automatically using CI!

User Permission Levels for CI Systems

User permission levels for CI systems are directly or indirectly related to the security of the underlying code repository. This relationship is natural because if you have permission to make a commit on the repository, then in effect you have permission to trigger any jobs based on that repository. Some CI systems such as CircleCI base their access permissions on the permission that users have on the underlying repository. Delegating access control and privileges to GitHub, for example, ensures that the CircleCI security model is simple but robust. Other CI systems such as Jenkins have access controls that are not directly related to the access controls on the repository.

Git itself does not handle access control. Instead, security is enforced by the Git host. And hosts such as GitLab, GitHub, and Bitbucket provide numerous layers of security over the repository itself.

The most basic level of access is simply whether a user can access the repository in any form. Additional layers of security exist to determine whether users can clone repositories, create branches, make commits on particular branches, and so on. On top of that, the CI system has a security model to determine which users can trigger jobs, see job logs, and see secret variables.

Most CI systems provide different security levels you can assign to users. For example, in GitLab, "developer" access is appropriate for most users since it allows them to use GitLab CI, but not to see or change secret values such as credentials.

To enable CI/CD, it's necessary to store credentials for the systems you connect to such as your Salesforce instances. Because your CI system has access to these credentials, it is important to secure the CI system just as you would secure access to the connected systems themselves. Only authorized users should have access to the repository, and you should monitor or periodically audit the list of who has access.

From a practical point of view, it's important to allow developers the ability to make commits on branches that trigger CI jobs. It's also important that they be able to see job logs so they can debug failures. But it's good practice to limit the visibility of the secret variables to just a few members of the team. This restriction does not interfere with the work or effectiveness of the team, but provides a layer of protection for these important credentials.

Creating Integration Users for Deployments

An entire team makes use of a single set of CI jobs, and to ensure consistent behavior, a single set of environment credentials are generally stored in the CI project and used by everyone. For example, if a team configures a job to automatically deploy metadata to a Salesforce org, which user's credentials are used for that deployment? It's often the case that a senior member of the team will configure the CI system and be tempted to use their own credentials for the deployment. That can lead to jobs failing if that person's user account is ever deactivated (think of the rate at which tech workers switch companies!). This can also leave that user's Salesforce credentials vulnerable or allow jobs to impersonate that user.

CI jobs that connect to other systems are integrations. As with any integration, you should consider the security of that integration. It's appropriate to create or use an integration user account for Salesforce deployments and tests and to use those credentials in your CI system. Until recently, deployments required "Modify All Data" privileges, but the new "Modify Metadata through Metadata API Functions"

permission provides a more secure alternative. Create a permission set that includes the permissions required to deploy metadata and configuration data to your orgs, and assign that permission set to an integration user. Then use those credentials in your CI process as opposed to the credentials for a specific individual.

Configuring CI/CD

As described earlier, every CI tool has their own way of configuring jobs. Nevertheless, there are certain common features to all CI systems. The main unit of CI configuration is a job. Each job has a name and then defines the source code used for that job, what triggers the job, what actions ("build steps") the job should take, as well as prebuild and postbuild actions and notifications. In other words, "what code are you working on?", "what action should be performed?", "when?", and "who/what should we notify?".

Groups of multiple jobs that have a single trigger but execute in a particular sequence are called a pipeline. Jobs may also be grouped into "projects" or some other kind of grouping for purposes of enabling or disabling access.

Some CI systems require you to log in to that system to configure jobs, but there are many reasons why it's helpful to use a CI system that allows you to store configuration as code.

Why Store CI Configuration As Code?

Storing CI job configuration as a configuration file inside the codebase is an example of "configuration as code." This ensures that any changes to this configuration are versioned and tracked and gives the team who controls the code the power to control their own CI processes without having to go through another group. Travis CI was the first CI tool to popularize this approach, but this approach is now used or supported by most CI tools. Each CI tool varies in the syntax used in this configuration file, but the files are generally short, easy to understand, and written in a human-readable markup language such as YAML or TOML.

Storing configuration in this way makes the configuration visible to everyone on the team, allows you to monitor changes to the configuration, and easily replicates it to other projects. It's even possible to autogenerate CI configuration files as part of project setup.

Storing Secrets and Using Environment Variables in CI Jobs

Environment variables are strings that are stored in memory and used to dynamically configure running applications. The most famous environment variable is `PATH`. If you type `echo $PATH` in Mac/Linux or `echo %PATH%` in Windows, you will see a list of all the paths that your system will check to find executables when you run a command from the command line.

Continuous integration processes are driven from source code stored in version control. But there are some things like passwords and tokens that should never be stored in version control. Environment variables are a perfect way to securely provide a continuous integration engine access to such "secret" configuration. This is also a recommended best practice for 12-factor apps (`https://12factor.net/config`).

There can be many types of configuration needed for a CI process to run. In general configuration for a CI process is best to be stored in the CI configuration file itself. But there are two main cases for using environment variables to store configuration:

1. Storing secrets like usernames, passwords, and tokens
2. Storing flags that you want to be able to change without changing the underlying config files, such as flags that enable/disable deployments

Environment Variables in the CI System

Most CI systems allow you to set environment variables in their configuration files or as secret variables injected on the fly. These variables are available in the job environment when it executes and can be referenced by all executed commands and scripts. Variables stored in configuration files should only be used to store nonsensitive project configuration.

Credentials and other secrets that are stored in the CI system's secret store are securely passed to the CI runner and made available as environment variables during a pipeline run. These values are not visible as plain text, and most CI systems will autodetect the presence of these secret values and obscure them if they appear in logs. This type of security is not impervious to being exposed, but nevertheless represents an important precaution. This is the recommended method for storing things like passwords, SSH keys, and credentials.

While you can set custom variables for use by your applications, CI systems typically include numerous built-in environment variables that you can also make use of. These CI variables provide information such as the commit message or ID, the URL of the CI system, the name of the branch, and so on. These CI-supplied variables supply useful information to allow you to write scripts that are more dynamic and context-aware.

Project-Specific Variables on Your Local System

If you need to store project-specific secrets as environment variables, you can put them in a local configuration file that is **not** checked into version control. Prior to the Salesforce CLI, it was common to store credentials for Salesforce orgs in a `build.properties` file that could be used by the Ant Migration Tool. The Salesforce CLI now has its own credential store for Salesforce orgs, but your project automation may still need to store some secret variables.

One common convention is to create a file called `.env` in the root of your project that contains any `key=value` pairs that you want to set. Node.js has a `dotenv` module that can be used to load these values, and tools such as Appirio DX provide out-of-the-box support for `.env` files. To load variables from this file manually on Mac or Unix systems, you can run the command `source .env` in a script.

Group-Level Configuration

Many CI systems allow you to organize multiple projects into groups and to specify configuration at the group level in addition to the project level. Specifying group-level configuration is important for two reasons. First it allows you to specify configuration that can be reused across many projects. If the configuration needs to change, it can be updated in one place to avoid the risk of individual projects getting out of sync. The other benefit of group-level configuration is to protect secrets that should not be visible to members of individual project teams. You might store the credentials for a production salesforce org at the group level and allow projects to make use of those credentials. But for users who don't have permissions at the group level, the credentials are not visible, providing a layer of added security.

Example CI/CD Configuration

Now that you're familiar with the concepts, let's review the automated jobs that you might configure as part of your Salesforce CI/CD workflow.

CI Jobs for Package Publishing

The core component of a Salesforce DX CI/CD workflow is package publishing. The "Branching for Package Publishing" section describes how a simple trunk-based strategy is sufficient for package publishing. Although feature branches can be used, they're not essential, and your initial CI configuration does not need to handle them.

Assuming your trunk branch is called `master`, your delivery pipeline will be triggered on every commit to `master`. First you can perform automated code checks such as static code analysis, then you can trigger unit test execution, then you can publish a new version of the package, and finally you can install it in a testing environment.

Continuous Delivery[21] suggests beginning by creating a "walking skeleton" that includes CI jobs for every step you intend to create, even if you've not yet determined the implementation details. In this case, define a CI pipeline (using whatever configuration your CI tool requires), specify that it should be triggered on `master`, and specify four jobs: "static code analysis," "unit testing," "package publishing," "package installation." The initial scripts for each of those jobs can be simple `echo Hello World` statements. If you're working with this CI tool for the first time, just getting this walking skeleton working may take you a few hours or more. Most CI tools have helpful tutorials to get you started.

Setting up a walking skeleton in this way is a type of agile development; within a short time, you have established a working CI process. You can now begin to refine the details and continue to refine and improve the process over the life of your project.

Salesforce DX jobs need to authorize with a Dev Hub and also with any target orgs. See the section on "Salesforce DX Org Authorizations" in Chapter 6: Environment Management for an explanation on how to use an Auth URL or a JWT token for org authorization. You will store these special strings as secrets in your CI system and use these secret environment variables to authorize the orgs before performing Salesforce DX commands.

[21] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

The most important job to define is the package publishing job, the heart of the workflow. Packages must first be created and their ID specified as an alias in the `sfdx-project.json` file. Then versions of that package can be published as the metadata evolves. See the Salesforce DX Developer Guide for detailed steps, but essentially you will be executing the command `sfdx force:package:version:create --wait 10`. This will create a new version of the default package defined in `sfdx-project.json`. If you are storing multiple packages in one repository, you will need to use the `--package` flag to publish versions of any nondefault package(s). The purpose of the `--wait` command is to force the package creation CI job to not terminate immediately but instead to wait for up to 10 minutes for this job to complete (you can adjust the duration as needed). Package version creation can be time-consuming, especially if you have specified managed package dependencies.

Each newly published package version has an ID. This ID begins with `04t` and is also known as a "subscriber package ID" because it can be used to install that package in another ("subscribing") org. When building this type of automation, it is best to request the output in JSON format by appending `--json` to the Salesforce DX commands. JSON can be read natively in most coding languages, especially JavaScript. You can also use the command-line tool `jq`[22] to allow for simple JSON parsing and reformatting. `jq` is one of several tools that I ensure are present in Docker images intended for use with Salesforce DX. See "Other Scripting Techniques" in Chapter 9: Deploying.

After extracting the subscriber package ID(s) from the results of the package version creation, you'll need a way to pass the ID(s) to the next job. Each job in a CI system is independent, and if you use Docker containers for your CI jobs, they each run in entirely different execution environments. CI tools provide mechanisms, however, for transferring information and files from one job to the next. In GitLab CI, for example, to retain files between jobs, you define "artifacts" containing particular files and folders. I will typically write any JSON outputs I need for other jobs into a file and include that in an artifact, along with the project's `.sfdx` folder.

The next job to configure is the "package installation" job. In this job you will unpack the subscriber package ID from the previous job and install the package you just created in a target org. This allows others on your team to review and test your updated package

[22] https://stedolan.github.io/jq/

in a live environment. First, use the `auth:sfdxurl:store` or `auth:jwt:grant` command to authorize the target org based on the Auth URL or JWT token you stored as a CI secret. You will then run a command like `sfdx force:package:install --package 04t... --targetusername yourAlias --wait 10`. The package ID is the subscriber package ID, the `targetusername` is the alias of the target org, and the `wait` parameter performs the same function as earlier.

Once those two jobs are in place, you have accomplished the deployment components of your CI system. Now you can define the first two test-related CI jobs based on the tools you're using for testing. You may wonder why I'm suggesting this order of defining the jobs, since it's not the linear order in which they'll eventually run. The purpose in doing this is to establish a functioning, end-to-end deployment process as early as possible, even if it's not perfect. DevOps implies a process of continuous improvement. But you can't improve your delivery pipeline if you don't actually have a delivery pipeline. The philosophy here is the classic approach to refactoring[23]: "make it work; make it right; make it fast." Package publishing and installation make this work. Static analysis and testing help make it right. There will certainly be opportunities to make this process faster once it's created, by being more selective about which tests run, and when packages are published and installed.

Running a static code analysis job depends on your having a static code analysis tool. Chapter 8: Quality and Testing explains the various options for static code analysis. Static analysis is most effective when presented as real-time feedback for developers in the form of "linting" in their IDE. But having a central static analysis job is useful to enforce team standards and to track metrics in a shared tool. Static analysis runs directly on your codebase. Most static analysis tools run locally, which in the case of a CI job means that the code is scanned within the CI job's execution environment. If you are running PMD, then there is no communication with an external static analysis server. You can export reports as downloadable artifacts in your CI system or (better yet) create a mechanism to upload and analyze the results in a central location. Other tools such as SonarQube communicate with a static analysis server to get the rule definitions, then run the scans locally, and report their results back to the static analysis server. This is a common architecture for static analysis tools that scales well even for hundreds of scans being run in parallel. When communicating with a static analysis server, you will need a token of some sort for authentication. You'll store this along with the other CI secrets.

[23] http://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast

Running unit tests is somewhat trickier, since this requires a Salesforce instance for those tests to run. Ideally developers should run tests on their development scratch org and ensure they pass before pushing changes for publication by the CI system. You can enforce the process of developers executing tests by adding that as a post-commit or pre-push hook in their local Git configuration. But synchronizing Git hooks across developers requires a tool like Husky,[24] and it's useful to have a mechanism to enforce test execution centrally.

Running unit tests requires a Salesforce environment that you can push your latest code to and run tests on. As mentioned earlier, it's important to establish a script that can be used to provision developer environments, review apps, and testing orgs. To create a test environment, simply run this script to create a new scratch org, install packages, and push source as you would when setting up a development environment. Then run your Apex tests (or an Apex Test Suite with a chosen subset of them) and pass or fail the job based on the test results. Typically, if a test fails, the command will return a nonzero status code, which Unix systems use to report failure. You can also set a test coverage threshold, report the test output in JSON, and then parse the coverage results to determine whether to pass or fail.

Once this entire system is set up and you have a working test and deployment pipeline, you've established the basic foundation for delivery automation. There are of course improvements you can make to make the process more effective or more efficient. If your scratch org creation process is time-consuming, you can consider whether you need to create a new testing scratch org each time. If not, you can store the org's access credentials as an artifact and access it repeatedly between jobs. To do this, simply persist the user level `~/.sfdx` folder as an artifact between jobs, and add logic to recreate the environment automatically if the scratch org has expired and can't be opened.

Perhaps you can add UI testing in the scratch org after Apex tests have run; or perhaps you want to run security scans or custom checks, for example, to ensure that custom fields all have descriptions in your XML metadata.

Do you need to publish every package every time? Perhaps you can dynamically determine which packages have changed and need updating by doing a Git diff between publish jobs. You can also use a tool like semantic release or monitor for changes to version numbers in the `sfdx-project.json` file to not publish new versions on every commit.

[24] https://github.com/typicode/husky

Do you want to enable automated or manual installation of this updated package in other environments? If so, you can enhance the package installation step or install into other environments as needed. A great tool to help with package installations is the Rootstock DX plugin rstk-sfdx-package-utils available on NPM.[25] `sfdx rstk:package:dependencies:install` checks your target org to see which package versions have already been installed and then installs any package versions that may be missing based on the list of package dependencies in sfdx-project.json.

To what degree you improve this workflow will depend on your team's priorities. The overarching goal of this process is to enable your team to be more effective by publishing small updates more frequently and reducing the frequency, severity, and duration of production failures. Use those goals to guide your priorities as you evolve this workflow.

CI/CD is what makes version control come to life and become a truly indispensable part of your workflow.

CI Jobs for Org-Level Management

Just as the package publishing workflow is based on the branching strategy for packages and brings it to life, so the CI process for org-level metadata management builds on the "branching for org-level configuration" described earlier and brings it to life.

As discussed in that section, the goal here is to provide centralized visibility and control over all orgs, allowing for both temporary and long-term differences between the orgs. Orgs have temporary differences due to features and fixes being gradually promoted and tested. Orgs have long-term differences related to integration endpoints, org-wide email addresses, and other org-specific configuration. While the goal of version control is to gain visibility into these similarities and differences, the goal of CI is to enforce control over the orgs.

As mentioned before, your goal should be to move the vast majority of your metadata into packages and use org-level management strictly for org-level configuration. As such, the static analysis and unit testing jobs have less importance in the org-level workflow, although it can be beneficial to add suites of integration tests that can be run on orgs to test the interplay between multiple packages.

As before, begin by creating a walking skeleton of the CI process that you want to enforce. This time, because you'll probably need to use multiple long-running branches

[25] www.npmjs.com/package/rstk-sfdx-package-utils

to manage configuration for your sandboxes, you'll be establishing multiple pipelines, each triggered by deployments to a particular branch.

Let's assume that you are managing SIT, Staging, and Production environments, and you are following the branching pattern described previously. You'll use the `master` branch to deploy to the staging environment, Git tags on the `master` branch to trigger the deployments to production, and a branch called `SIT` to manage the metadata in that environment. If you're using GitLab CI, a walking skeleton might look like the one in Listing 7-3.

Listing 7-3. Sample .gitlab-ci.yml file for managing org-level metadata publishing

```
image: myCompany/salesforceDXimage:latest

stages:
  - deploy

deploy_to_SIT:
  stage: deploy
  script:
    - echo Hello World
  only:
    - /^SIT/

deploy_to_staging:
  stage: deploy
  script:
    - echo Hello World
  only:
    - master

deploy_to_production:
  stage: deploy
  script:
    - echo Hello World
  only:
   - tags
   - /^v[0-9.]+$/
  when: manual
```

Because we are managing multiple branches, there are multiple CI pipelines defined even when there's only one job in each pipeline. It is the `only` property that establishes these three jobs as distinct pipelines. We are indicating that this job will only be triggered when changes are made to a particular branch or tag. In GitLab, pipeline jobs are calculated dynamically based on trigger conditions like `only`, `except`, and `when`. We can add additional jobs to a pipeline by using the same trigger conditions on multiple jobs.

Begin by defining the specific scripts to run to deploy updates to the SIT environment. Once you determine the appropriate pattern for that deployment, you can replicate it in the job configuration for the other environments.

The basic pattern for each org is to install any package updates, build and deploy the metadata specific to that org, build and deploy configuration data specific to that org, and then execute automated acceptance tests. While there are multiple ways to divide jobs, you might group the middle two (closely related) processes and have jobs called `update_packages`, `update_configuration`, and `acceptance_tests`.

Managing the installed package versions at the org level provides central visibility on the differences between orgs. But since package installation can also be done using the delivery pipeline for packages, we can begin by defining the `update_configuration` job.

The section on "Managing Org Differences" in Chapter 9: Deploying provides more details on a folder structure for tracking metadata differences between each org. But in short, it's helpful to separate the metadata that is common to all orgs from metadata that is unique to particular orgs. When you perform deployments, that metadata will need to be combined in a temporary folder from which you create a single deployment. This takes advantage of Salesforce's transaction processing which allows the entire deployment to be rolled back if any part of the metadata fails to deploy.

Configuration data can be combined in a similar way, giving precedence to org-specific configurations over the default configurations. That configuration data can then be loaded in the target org using the data loading methods provided by the Salesforce CLI.

With these processes, there's obviously room for optimization once they're established. As described earlier, I've typically used Git tags to mark a successful deployment and Git diffs to determine what metadata or data has changed since the last deployment. This information allows us to fill a temporary deployment folder with only the metadata and data that needs to be updated, making for a faster deployment and update.

The acceptance testing process is similar to the unit testing process described earlier, with several differences. First, there is no need to create a testing environment, since these tests will run on sandboxes and production. Second, these acceptance tests should be more comprehensive and wide ranging than the unit tests. While there is no need to repeat the unit tests, these acceptance tests should exercise key user workflows, especially critical business process calculations, to ensure that there are no regression failures.

An acceptance test is not different in nature from a unit test, it simply involves tests across multiple parts of the system and may be longer running as well. See the section on "Automated Functional Testing" in Chapter 8: Quality and Testing.

When defining the scripts to manage package installation at the org level, you'll approach this in three phases. In your first iteration, simply (re)install all the packages for the org each time the job runs. That gives you a way to push out package upgrades from your CI system, but is clearly inefficient unless you have a tiny org. In your second iteration, determine currently installed packages and update only the ones which differ from your configuration. This creates a highly efficient package upgrade process. Finally, build a mechanism to connect the repositories that manage your package publishing with the repository for org-level metadata, such that publishing a new package version triggers a package upgrade process in your org-level repository. That kind of multiproject pipeline allows each repository to be simple and effective while still orchestrating a larger process.

Salesforce DX does not contain explicit guidelines for managing packages at the org level. But the `sfdx-project.json` gives a format for listing package dependencies that can be extended for use in managing org-level package "dependencies." The folders used to hold org-specific metadata can be listed as "packages" in `sfdx-project.json`. The package versions to be installed in that org (both external managed packages and unlocked packages from your own team) can be listed as "dependencies" of that package. Any automated methods to install dependent packages in a scratch org can then be extended to manage package installation/update in a sandbox or production org.

Determining the package versions currently installed in an org is an important way to avoid spending time with redundant installations of packages that are already up to date. Salesforce provides the command `sfdx force:package:installed:list -u yourOrgAlias` as a way to query those packages. By using the `--json` flag, you can export an easy-to-parse list of those packages which you can then compare with the package

versions held in version control. Again, the Rootstock package installer[26] can be used to automate this entire process.

Once you establish a reliable method to install packages at the org level, you should remove the capability to install packages from the package publishing workflow. This prevents the risk of the two CI pipelines installing different versions of the packages.

Instead, you should establish a mechanism for package pipelines to generate commits or merge requests onto the org-level pipeline. When a new package version is published, it can push a change to the org-level repository to share the new package version or ID. The most effective mechanism to do this will depend on your CI system and your team's chosen workflow. GitLab has a robust API that can be used to make updates to files and generate commits, branches, and merge requests. If your CI system does not offer such an API, you can trigger a CI job that queries the Dev Hub for the latest versions of particular packages and then writes the latest version back to the repository. Note that CI jobs don't normally write to their own repository so you may need to add a token for your repository as a secret variable and use that to perform a push from the CI job back to the repository.

Any detailed scripts you create should be stored as files in your repository, outside of the CI configuration file, allowing the CI configuration to remain concise and readable. Nevertheless, if you find you're accumulating repetitive blocks of YAML, you can use anchors to indicate blocks of YAML that should be repeated in the configuration. Anchors are a standard part of YAML syntax, but you should confirm whether your CI system supports them.

Instead of a block like the one shown in Listing 7-4, you can use anchors to avoid repetition as shown in Listing 7-5. While the difference here is small, anchors can be a helpful way of keeping your configuration readable as it grows.

Listing 7-4. Sample .gitlab-ci.yml YAML file with repetitive blocks

```
image: myCompany/salesforceDXimage:latest

stages:
  - install
  - deploy
  - test
```

[26] www.npmjs.com/package/rstk-sfdx-package-utils

```
update_packages_in_SIT:
  stage: install
  script:
    - ./scripts/getInstalledPackages.sh $TARGET_ORG
    - ./scripts/updatePackages.sh $TARGET_ORG
  variables:
    - TARGET_ORG: SIT
  only:
    - /^SIT/

update_packages_in_staging:
  stage: install
  script:
    - ./scripts/getInstalledPackages.sh $TARGET_ORG
    - ./scripts/updatePackages.sh $TARGET_ORG
  variables:
    - TARGET_ORG: staging
  only:
    - master

update_packages_in_production:
  stage: install
  script:
    - ./scripts/getInstalledPackages.sh $TARGET_ORG
    - ./scripts/updatePackages.sh $TARGET_ORG
  variables:
    - TARGET_ORG: production
  only:
   - tags
   - /^v[0-9.]+$/
  when: manual
```

Listing 7-5. Sample .gitlab-ci.yml file using YAML anchors

```
image: myCompany/salesforceDXimage:latest

stages:
  - install
  - deploy
  - Test

.fake_job:
  <<: **&update_packages**
    stage: install
    script:
      - ./scripts/getInstalledPackages.sh $TARGET_ORG
      - ./scripts/updatePackages.sh $TARGET_ORG

update_packages_in_SIT:
  <<: ***update_packages**
  variables:
    - TARGET_ORG: SIT
  only:
    - /^SIT/

update_packages_in_staging:
  <<: ***update_packages**
  variables:
    - TARGET_ORG: staging
  only:
    - master

update_packages_in_production:
  <<: ***update_packages**
  variables:
    - TARGET_ORG: production
  only:
   - tags
   - /^v[0-9.]+$/
  when: manual
```

Summary

The delivery pipeline is the mechanism to deliver features and fixes safely and quickly from development to production. The foundation for this is version control, and your version control branching strategy is used to balance the needs for freedom, control, and ease in your deployments. CI automation tools are driven from version control, and are the engine that enables deployments, tests, and any metadata transformations that may be needed.

In the next chapter, we'll look in detail at testing, the aspect of the delivery pipeline that fulfills the need for safety and reliability.

CHAPTER 8

Quality and Testing

Quality refers to the degree to which software does what it's supposed to do, and is built in a way that is structurally sound. Testing refers to checking and giving feedback on software quality. Testing can assess functional quality: does it do what it's supposed to do and not do things it shouldn't? Testing can also assess structural quality: is it reliable, secure, maintainable, performant, and appropriately sized?

Some amount of everyone's work and customizations will be thrown away or never used. If your work is lucky enough to be used in production, it will always be tested. Your work will either be tested intentionally prior to release, or it will be tested implicitly when it's used or updated. When end users intentionally review and give feedback on functionality that is pending release, this is called "user acceptance testing." But sometimes unsuspecting end users are exposed to functionality that has not been tested by any other means. In that case, the "testing" is inadvertent, uses real business data, and can lead to confusion, problems, and ill-will.

The purpose of this chapter is to explain key concepts of Salesforce software quality, share the many available testing mechanisms, and encourage practices that increase the reliability of your production systems. Although I frequently reference "code" quality, Salesforce customizations involve an interplay of code and non-code configuration. In this chapter, the term "code" is mostly used as concise shorthand for any Salesforce customization.

Understanding Code Quality

Quality is subjective, but needs to be considered from the perspective of both end users and the technical team that creates and maintains the entire system. End users are concerned mostly with whether code functions as it should, whereas it is the role of the technical team to ensure that the code is maintainable, secure, performant, testable, and

© Andrew Davis 2019
A. Davis, *Mastering Salesforce DevOps*, https://doi.org/10.1007/978-1-4842-5473-8_8

so on. Although end users aren't concerned with the underlying implementation, flaws in the code that lead to slow performance, data loss, and exploits can quickly become massive issues that capture the attention of the business, customers, and even the media.

Quality code can be seen as code that meets the current and *potential* needs of the customer. This is by definition a very challenging problem. Quality is not something you can perfect; but it's often something you can improve.

Functional, Structural, and Process Quality

As shown in Figure 8-1, software quality can be divided into functional quality, structural (or nonfunctional) quality, and process quality.

Functional quality refers to whether the software functions as it should and meets the stated needs of the customer. In agile terms, code that fulfills the purpose of the story and meets the acceptance criteria can be said to have functional quality. One of the benefits of iterative development is that the sooner you can put working software in front of end users, the sooner you can validate that it meets their actual needs. If end users are not satisfied, further development cycles can focus on increasing the functional quality in their eyes.

Structural quality, also known as nonfunctional quality, deals with the quality of the underlying implementation. Well-crafted software will function *reliably*, day in and day out, even as other aspects of the system evolve. The evolution of any IT system is inevitable—it is a living thing—and so the *maintainability* of code is also crucially important. It should be possible to make small changes to the system in a small amount of time, without undue effort spent deciphering the meaning of poorly named, poorly commented, rambling "spaghetti code." And as the user base continues to grow, functionality continues to expand, and data accumulates, the application should *perform* consistently at scale. Software is designed with certain users and use cases in mind. But how will it behave when unintended users do things to the system that were never considered in its design? The IT world exists under the shadow of countless threats. Any system exposed to the Internet will immediately be exposed to thousands of automated attacks from bots built to identify and exploit common vulnerabilities. The *security* of your application is thus a critical hidden factor that must be considered.

In all of this, the *size* of your code, both its scope and the LOC (lines of code) count, has a great impact. For example, a design that served the application well when it was just beginning may be entirely unsuitable once the application has grown to enterprise levels.

Process quality refers to the quality of the development process: whether it supports the development of quality software in an efficient and reliable way. Much of the rest of this book focuses on process quality, such as the use of version control, continuous integration, and automated testing. Process quality also includes the process of gathering and tracking requirements and coordinating the work of the development team.

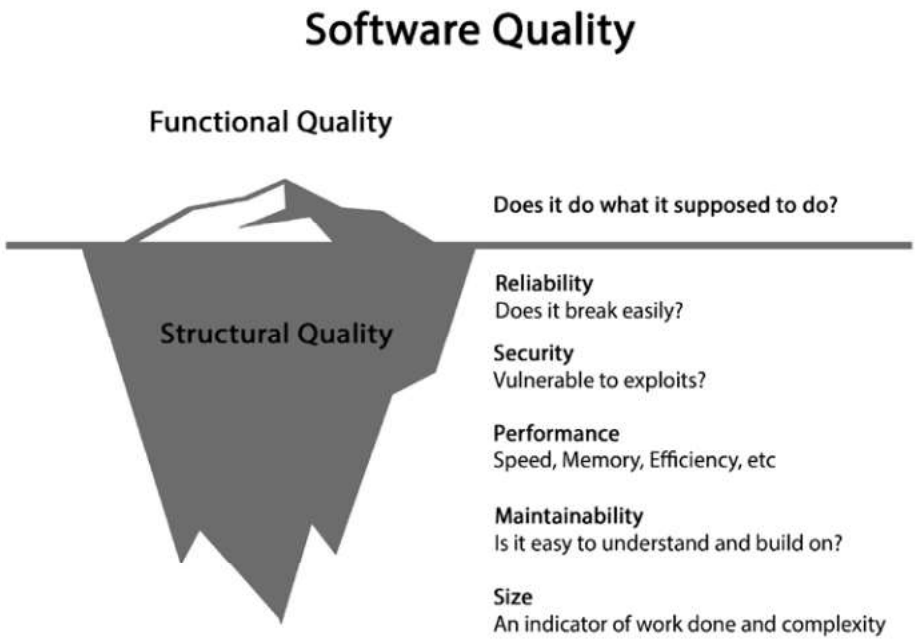

Figure 8-1. *Code quality has many hidden levels*

Understanding Structural Quality

It is important for development teams to understand structural quality in more detail, so they can understand how to improve and ensure it. Different sources use different terms to describe the aspects of software quality, but the following divisions cover most concerns.[1]

[1] https://en.wikipedia.org/wiki/Software_quality

Reliability

Reliability refers to the ability of your code to perform consistently, every time it's executed, in spite of variations in inputs, changes to other systems, and ongoing updates to functionality. A reliable system "just works": no 500 errors, unhandled exceptions, and the like. This implies that the system has been tested with inputs including edge cases such as null or missing values, values approaching and exceeding size limits, and so on. Reliability implies that your design is not brittle and dependent on underlying data and systems that might change due to factors outside your control. If a key piece of configuration data is suddenly absent, your application should generate a meaningful error on the backend and fail gracefully for users.

Many DevOps practices such as "chaos engineering" are focused on building reliable distributed systems at scale. Reliability at scale implies that systems should continue to function even if individual nodes periodically go offline.

One key to ensuring reliable code is to practice test-driven development (TDD) or its offshoots such as behavior-driven development (BDD). Underpinning each acceptance criteria for your application with an automated test that ensures this functionality allows you to run regression tests after every significant change to your application. This provides quick visibility into any failures that might compromise functionality. Writing a rich set of automated tests depends on your code being testable. To be testable, code needs to be modular. And testing code that depends on external systems requires there to be a layer of abstraction so that a "test double" can be used instead of actually contacting the external system. In this way, the process of writing tests creates a natural pressure toward good coding practices.

Maintainability

The people who maintain software are rarely the ones who originally wrote it. If a developer revisits code they wrote 6 months ago, it might look as unfamiliar as code written by another person. Maintainability means that software is written in such a way that the purpose of each method is easy to understand and that small changes can be made easily without huge effort and without fear of breaking the application. Clear naming of variables, classes, and methods is critical. If it takes you 5 minutes to figure out what a small method does, take 1 minute to give it a more helpful name that clearly describes its purpose. As Figure 8-2 indicates, naming things is hard. Invest time in choosing self-explanatory names.

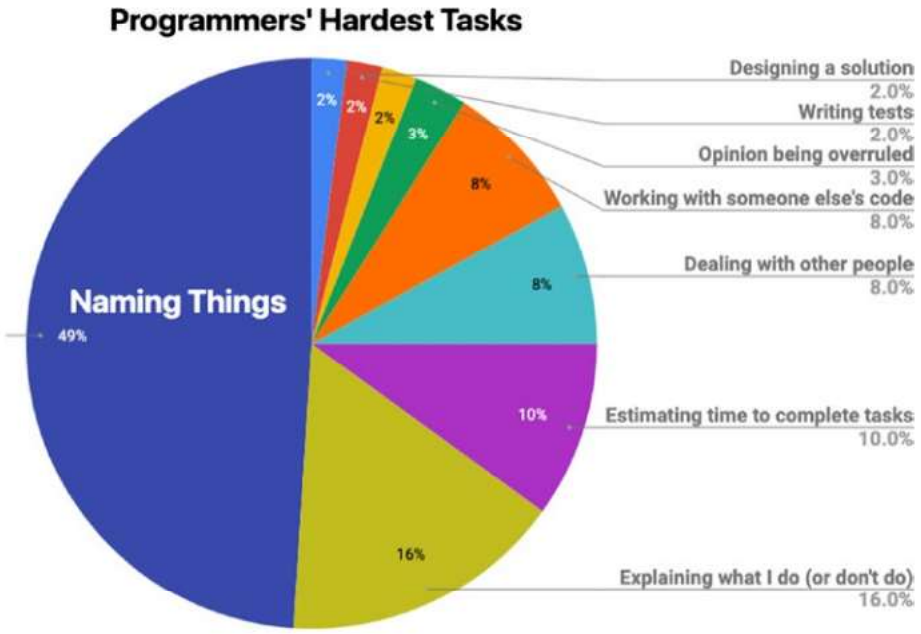

Figure 8-2. *Naming things is hard*[2]

Another important aspect of maintainability is reducing the "cognitive complexity" of the code by reducing the number of if statements, loops, and other logic in a single block of code. This is also known as "cyclomatic complexity": the number of different logical paths a section of code could follow.

Reducing the sizes of classes and methods is a key way to ensure maintainability. The "single responsibility principle" states that every piece of code should have just one purpose. A method that does two things should be divided into two methods with very clear names. Adding abstraction in this way makes code quick and easy to understand, without requiring maintainers to spend hours deciphering complex logic.

Code comments are very important to explain complex pieces of logic and to otherwise clarify the code. But wherever possible use simple logic, clear names, and short methods instead of writing comments. A standard to strive for is "self-documenting code," where the names are so clearly chosen, and methods are so brief and well structured that even a nonprogrammer could understand its purpose just by reading it.

[2] www.itworld.com/article/2833265/don-t-go-into-programming-if-you-don-t-have-a-good-thesaurus.html

There are many other design principles that contribute to maintainable code. Larger codebases require enterprise design patterns to ensuring maintainability. The book *Clean Code* by "Uncle" Bob Martin[3] is a classic recommended for all programmers.

Performance

Performance implies that your code will function at an acceptable level even as the number of users, parallel processes, and data increase. Performance testing in a "production-like" environment is a key stage of software development. And capacity planning—anticipating both best case and worst case rates of growth—is important to ensure you don't encounter a scaling crisis a year or two into production.

Salesforce takes care of the performance and scalability challenges of the underlying platform. But the performance of your customizations is the responsibility of your own team.

Security

Security vulnerabilities are arguably the scariest flaws in software. Nevertheless, security analysis and remediation are among the most neglected aspects of quality analysis. Developers may be oblivious to security threats or simply hopeful that they're not an issue. Security analysis can also be a specialized skill, neglected during developers' basic training. For these reasons, having automated tools that can identify security vulnerabilities and recommend resolutions is key. This is especially important for any system that faces the public Internet or carries sensitive information such as financial data. Web administrators can assure you that the Internet is a hostile place, with new servers receiving automated scans and attacks within minutes of going online.

Size

Size is a consideration in structural quality simply because the larger the application becomes (both in terms of code and functionality), the more challenging the preceding issues become. As the size of an application increases, its *attack surface* increases, meaning that there are more possible vulnerabilities. The principle of *security asymmetry* says that it's always harder to defend than to attack, since an attacker only needs to find one way in, whereas the defender needs to defend all possible points of entry.

[3]Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (Prentice Hall, 2008).

Understanding Process Quality

Whereas functional and structural quality relate to *what* is being built, process quality refers to *how* you are building.

There are two main aspects of process quality: does your process lend itself to creating a quality product and is the process itself safe, sustainable, efficient, and so on?

The main focus of this book is to suggest technical processes that lead to higher-quality work as well as a more sustainable and efficient process, so the entire book can be seen as falling into this topic. A comprehensive discussion of process quality would also touch on many business and human aspects and is beyond the scope of this book.

Testing to Ensure Quality

Many factors ensure quality. As the saying goes, "quality is everyone's responsibility." It's also said that quality is far too important to be left to the QA team. Developer training and skill is of course important, as is a manual testing process. But between development and manual testing are automated systems that can be put in place to provide fast feedback to developers, early warning of issues, and reduced burden on manual testers.

Many aspects of this book deal with development process improvements such as the use of version control, continuous integration, and modular architecture. But what processes and systems can be put in place to help enforce and ensure quality in the development process?

Why Test?

The purpose of testing is to protect current and future users from unreliable systems. Another way of expressing this is that testing helps build quality into the work you're delivering.

As shared in Chapter 3: DevOps, there are five metrics you can use as key performance indicators (KPIs) of your software delivery process. While the first two deal with speed, the latter three deal with system reliability: deployment failure rate, mean time to recover, and uptime. Historically, most teams assumed there is a tradeoff between speed and reliability, or between innovation and retaining users' trust. But the *State of DevOps Reports* validate that high-performing DevOps teams excel in both speed and reliability.

The only way to deliver innovation to production more quickly without sacrificing reliability is to build quality into your delivery process. W. Edwards Deming, widely credited with transforming industry in Japan and later the United States, offered a 14-point synopsis of how businesses can improve their operations.

> *Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place.*[4]
>
> —Point 3 of Deming's 14 points

Deming's work is credited with instigating the Total Quality Management movement in business. The basic idea is simple: as shown in Figure 8-3, the earlier issues are caught, the less expensive they are to resolve. Finding bugs through manual QA is time-consuming and expensive compared to using automated methods. But that cost is far less than the cost of end users finding and reporting these bugs in a system they assumed was reliable.

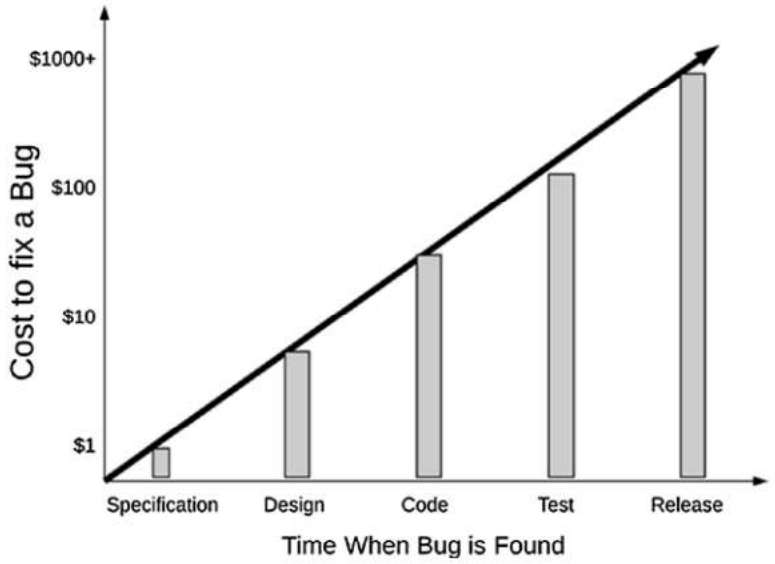

Figure 8-3. *The cost of fixing bugs increases the later in the process they are found*

[4]Deming, W E. Out of the Crisis. Cambridge, Mass: Massachusetts Institute of Technology, Center for Advanced Engineering Study, 1986. Print.

This also gives rise to the exhortation to "shift left." As shown in Figure 8-4, this means to put increasing emphasis into ensuring that the original design, architecture, and coding are of high quality, rather than attempting to catch most bugs in a manual testing phase. Manual QA will never be 100% reliable. In addition to the high costs of manual testing, if nothing is done to address the quality of the inputs (the original development), quality issues will continue to slip through no matter how much you adjust your manual testing methods. Deming often used the "Red Bead Experiment"[5] to provide a visual demonstration of this concept. If an incoming collection of white beads is contaminated by some amount of red beads, those red beads will continue to pollute downstream processes despite the best efforts of workers and management. The only way to ensure that the final product is reliable is to ensure that the inputs are reliable, which points to the importance of giving developers fast, high-quality feedback.

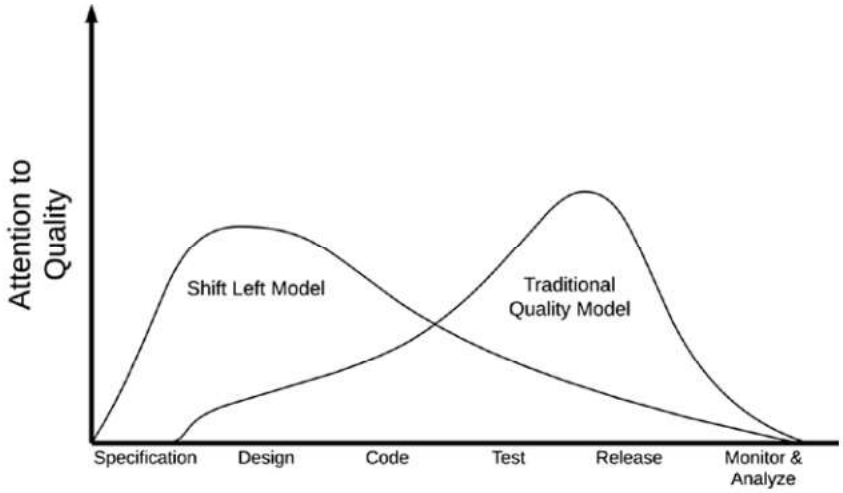

Figure 8-4. *"Shifting left" reduces overall costs by addressing quality issues earlier in the development lifecycle*

[5] www.youtube.com/watch?v=geiC4UgpDyw

What to Test?

In general, you need to test the business-critical customizations you make on the Salesforce platform. With extremely rare exceptions, you don't need to test Salesforce itself; and you don't need to test trivial customizations like page layouts. But you can and should test any places you use complex logic (including complex validation rules and workflow rules) to ensure that your own logic is correct. And you should have automated regression tests in place to ensure that critical business functionality remains correct, especially around the systems that help your company to make money.

If your employee idea board goes down for 8 hours, it may be an annoyance to employees, but it's not likely to cost your company any money. If your sales people are unable to work for 8 hours, or your eCommerce site goes offline for that period, you will have sacrificed a portion of your annual sales. In short, not every system has equal value to the company, and thus not every test has equal protective value.

You should prioritize tests according to the production value of the system under test.

Testing Terminology

Different sources use different names to describe different kinds of tests. And just as with "different kinds of mammal" and "different kinds of emotion," the distinctions are not always clear. Tests can be distinguished based on whether they test the behavior of the system or its characteristics (functional or nonfunctional tests), how they interact with the system (code tests, API tests, or UI tests), what level of the system they test (unit, component, or integration tests), how they are performed (manual, automated, or continuous), and when they should be run (commit stage tests or acceptance tests). This section briefly introduces those distinctions in hopes of demystifying them.

Don't waste time feeling bad if you're confused about testing terminology. The software testing community has created an amazing variety of terms to describe types of tests. Perhaps this is because they have been forced to do so much tedious, manual testing that their creativity needed to find an outlet. I'd be complicit in this linguistic crime if I shared them all, but I refer you to `www.softwaretestinghelp.com` if you want to satisfy your curiosity about how monkey testing differs from gorilla testing.[6]

[6] `www.softwaretestinghelp.com/types-of-software-testing/`

Functional and Nonfunctional Tests

There are different ways to look at a system: what it does, and what it's made of. Functional tests judge what something does. For example, if a bit of logic is supposed to apply price discounts for customers with an annual contract value above 1 million dollars, you can write functional tests to ensure that logic is correct. Nonfunctional tests look at other aspects of the system, like how long it takes to run, whether it suffers from security or coding style flaws, and so on. This is discussed in detail in the section "Functional, Structural, and Process Quality."

Code, API, and UI Tests

The only built-in testing tool that Salesforce provides is the ability to run tests written in Apex. For this reason, when people refer to "Salesforce testing," they are often referring to tests written in Apex that test the behavior of other Apex classes or triggers. It's helpful to take a broader view of this though and distinguish that there are three ways that a test might be run: via code, via an API, or via UI tests.

Code-based tests include Apex tests as well as JavaScript tests of Lightning Components and the JavaScript used in VisualForce. API-based tests are those that use Salesforce's APIs to test behavior. API tests become quite important when exercising integrations with other systems. MuleSoft's MUnit capabilities[7] are integration tests that can be used to validate behavior using an API. There's a gray area between code and API in cases where tests are written in languages such as Ruby or Python to test the behavior of Salesforce, since those tests generally use the Salesforce API to test.

UI tests are what you do when you log in to Salesforce to observe and check its behavior. Manual QA and UAT are almost always done in this way. UI testing can also be done in an automated way, using tools such as Selenium, Puppeteer, Provar, or Tosca.

Unit, Component, and Integration Tests

It's very common to use a threefold distinction between unit, component, and integration tests. Unit tests are those that exercise only one unit of code, such as a single method. Component tests are those that exercise multiple units of code or configuration, such as a test on a Trigger that also executes Trigger handler classes, or other business logic. Integration tests are those that exercise an entire integrated system, cutting across

[7] www.mulesoft.com/platform/munit-integration-testing

multiple Salesforce components or maybe even across other systems. Apex can be used to write unit tests or component tests that also test workflow rules, processes, and flows. But it can't make HTTP callouts and has CPU, memory, and execution time limits that limit your ability to write true integration tests in Apex.

This threefold distinction is also the most common source of terminology confusion. "Integration test" can be used to mean different things. Salesforce developers often refer to any Apex-based test as a "unit test" even if it cuts across many components. I use the term "unit test" exclusively for code-based tests, but I've (unfortunately) heard people refer to narrowly scoped manual tests as unit tests. Google did a nice job of cutting through such confusion by dividing tests simply into small, medium, and large.[8]

Manual, Automated, and Continuous Testing

Central to the points made in this book is this threefold distinction in terms of how tests are performed. Manual tests are those where humans perform actions and check the output. Manual testing is the default behavior for traditional quality testers, for user acceptance tests, and even for programmers and tech leads wishing to validate a system. I've heard it said that you can't automate something that you can't do manually, so manual testing may also be a precursor to other kinds of automated testing.

The benefit of manual testing is that much of it can be done by relatively low-skilled workers, it doesn't require any specialized tools, and it doesn't require much planning or architectural consideration. The challenges with manual testing are that it's relatively slow (minutes or hours instead of seconds), it's error-prone, it's tedious (if you have to perform regression tests repeatedly), it requires people to be available (which may cause delays), and it's expensive (even offshore resources get paid more than computers). The challenges with manual testing should drive you to reduce reliance on it and to upskill your manual testers to help write automated tests.

Automated tests are those that can be done by computers. The benefits of automated tests are in speed, reliability, and cost. Those benefits change the economics of testing and open the door to performing automated tests far more frequently and across a diversity of platforms. Platforms like SauceLabs and BrowserStack revolutionized web testing by allowing companies to test their web sites or applications across a myriad of devices and browsers in parallel, uncovering issues in minutes that might otherwise have taken weeks to discover.

[8] https://testing.googleblog.com/2010/12/test-sizes.html

Continuous testing refers to running automated tests every time code is changed on a particular branch in version control. Continuous testing is in fact an aspect of continuous integration/delivery, since it provides a way to ensure the reliability of a system that is undergoing rapid change. Not all automated testing tools support continuous testing. For example, Tricentis Tosca provides a nice set of tools to build UI tests for Salesforce. But it can only run on Windows-based systems, and so it can be a struggle to use it for initiating automated tests from a Linux-based CI system.

Commit-Stage Tests and Acceptance Tests

Finally, tests can also be distinguished in terms of the point in the development lifecycle when they are run. This distinction is made in the *Continuous Delivery* book and is relevant when planning your approach to testing.

The basic idea is that some tests should run every time you make a commit in version control (continuous testing), but that it's important that those tests can complete in 5 minutes or less. The reason is that unless developers get fast feedback from those tests, they are likely to ignore them. It also makes it harder to recognize if a test failure is associated with a particular change if several hours have elapsed since the change was made.

In addition to this fast-running subset of tests, you need a robust and comprehensive set of tests that can be run before code is finally released. Acceptance tests refer to the broader set of tests designed to ensure that your work meets requirements before it's released. These tests are generally functional tests that ensure your code functions properly. They're often accompanied by different nonfunctional tests that also help ensure quality.

Thus commit-stage tests represent a small and fast-running subset of acceptance tests. Commit-stage tests rarely include UI tests, since those tend to take longer to run. In the Apex world, it's beneficial to create one or more test suites that contain certain critical test classes you want your developers to run regularly. The point of these commit-stage tests is to provide an early warning to developers if they break something and to provide it immediately to minimize time spent diagnosing the cause.

The remainder of this chapter is divided into two sections: "Fast Tests for Developers" and "Comprehensive Tests." "Fast tests" is the simplest term I could think of to describe commit-stage testing, and hopefully the meaning is unambiguous. "Comprehensive tests" is the simplest term I could think of to describe acceptance tests, which will accumulate over time to represent the widest range of test cases, with much less regard for how long they take to run.

Yes, I just added two new names to the overflowing lexicon of testing.

Test Engines

The different types of test discussed later require different "test engines." For Apex tests, the Apex test runner is a native capability of Salesforce. For manual tests, the "test engine" is the person performing that test. For other types of test, you'll typically need a tool to help with this process.

In each of the following test types, you'll find discussion about which tools can help support that process. Some tools like JMeter are specific to one type of testing, in this case performance testing. Other tools, like static analysis tools, can help across different stages of the test lifecycle.

Test Environments

Different types of testing bring different demands for test environments. The general strategy to support these demands is to "shift left" as much as possible and to try to batch multiple different demands into a small number of environments as illustrated in Figure 8-4. Details about the environments needed for the various types of tests are shared along with the description of those tests later.

As described in the following, some kinds of testing require long-lived, fully integrated sandboxes, whereas other types can be performed in scratch orgs or disposable developer sandboxes. Long-lived sandboxes that need to be integrated with external systems and require manual setup are probably best to be created manually from the **Environments ➤ Sandboxes** screen in your production org. If you find yourself regularly needing to refresh them, however, you may consider investing in automating their creation.

Short-lived testing environments should be created and destroyed automatically as part of your CI process, using the Salesforce CLI and the same set of scripts you use to provision developer environments. The Salesforce CLI now also allows you to create and log in to sandboxes, so this process can be used to automate the refresh of testing sandboxes.

Test Data Management

The vast majority of test types require data. "Given certain input data, do we receive the correct outputs?" Some types of test require minimal data, others require massive amounts of data. Managing test data is an integral part of your test setup. Details on providing appropriate data are integrated with the following descriptions of each test type.

Fast Tests for Developers

This section and the following "Comprehensive Tests" section constitute a twofold division of all the different types of tests. Fast tests refer to tests that should be run before and/or after every commit. The formal name for this is commit-stage testing, which consists of code analysis and unit tests.

Coding is hard. Managing a team of distributed programmers, enforcing style guidelines, and performing code reviews are even harder. Fortunately, there are automated tools that can help enforce and remind developers of standards and best practices for each language.

Code quality analysis is an automated system for providing code feedback and a helpful mechanism to improve code quality. It's like spelling and grammar check for your code. Code quality maintenance and improvement requires attention and focus throughout a project's lifecycle. Issues with code quality, such as poorly designed or poorly documented code, will accumulate easily if left unchecked. These issues are known as **technical debt**, and if left to grow, they make software maintenance increasingly difficult, time-consuming, and risky. In the same way that one might deal with financial debt, the key to mitigating technical debt is to acknowledge and address quality risks or concerns as early as possible in the development process—not to let them accumulate.

Throughout the remainder of the chapter, we introduce several types of test and how you can run them. In each case, we address the following questions:

- What is this test? Why do you need it?
- When are these tests triggered?
- What environment(s) does the test run in?
- What data do you need to run this test?
- How do you create these tests?
- What else should you consider regarding these tests?
- What happens with the results of these tests?

Hopefully this will provide a consistent guide as you gradually layer in different types of testing.

Static Analysis—Linting

Static analysis is an automated analysis of the source code to identify possible structural faults in performance, style, or security. A simple example is measuring the length of each class or method and marking long classes or methods as possible maintainability problems. The benefit of static analysis is that it can be run quickly and inexpensively, as often as needed, and so can provide feedback faster than any other type of test.

Static analysis can be performed as one stage in a CI process, but is most effective when it's done in real time in the developer's code editor. This kind of immediate feedback is called *linting* and is similar to spell checking or grammar checking. This provides real-time feedback on code right inside the code editor … there's no better method to ensure developers see and act on feedback.

Linting is a great example of shifting left, since it's done during the coding process itself as opposed to waiting for the CI/CD server. Whereas unit tests are used to confirm the specific behavior of your code, static analysis checks to ensure that it complies with general rules such as "methods should not have too many parameters."

How to Run Linting

By definition, linting runs in the developer's code editor in real time. Linting is language-specific. For JavaScript there are many linting tools available, but ESLint has come to be the dominant choice. ESLint provides general feedback on JavaScript style. It offers a set of recommended rules, but can also be configured to enable or disable additional rules.[9] Additional ESLint rules have been written for both (Aura) Lightning Components[10] and Lightning Web Components,[11] making ESLint the clear choice for JavaScript linting in Salesforce.

I am aware of two linting solutions for Apex code: ApexPMD and SonarLint. PMD is the most popular static analysis tool for Apex (partly because it's free). Chuck Jonas wrote a well-maintained PMD extension for VS Code.[12] PMD is also integrated into the commercial IDEs, Illuminated Cloud and The Welkin Suite.

[9] https://eslint.org/docs/rules/
[10] https://github.com/forcedotcom/eslint-plugin-aura
[11] https://github.com/salesforce/eslint-plugin-lwc
[12] https://marketplace.visualstudio.com/items?itemName=chuckjonas.apex-pmd