Deployments

Deployment Best Practices. Focuses on CI/CD as this is the current best practice.

Introduction - Why are we even doing it like this

Salesforce deployments are essential for managing and evolving Salesforce environments, especially in a consulting company setting. There are several methods for deploying metadata between organizations, including Change Sets, the Metadata API, and the Salesforce Command Line Interface (CLI). Each method has its unique advantages, but the introduction of Salesforce DX (SFDX) has revolutionized the process, making SFDX-based deployments the standard for the future.

> The main reasons are because it is easy to deploy, and easy to revert to a prior version of anything you deploy as well - proper CI/CD depends on GIT being used, which ensures that everything you do can be rolled back in case of bugs.

A table of deployment methods with advantages and disadvantages

Deployment Method	Advantages	Disadvantages
Change Sets	- Easy to use with a graphical interface - No additional setup required	- Limited to connected orgs - Manual and time-consuming - No version control - Can be done ad-hoc
Metadata API	- Supports complex deployments - Can be automated - Broad coverage	- Requires programming knowledge - Steeper learning curve
Salesforce CLI (SFDX)	- Advanced automation - Supports modern DevOps practices - Version control	- Steeper learning curve - Initial setup and configuration required - Requires trained staff to maintain

Deployment Method	Advantages	Disadvantages
Third-Party Tools	- User-friendly interfaces - Advanced features and integrations	- Additional costs - May have proprietary limitations

> Despite the complexity inherent in SFDX-based deployments, the benefits are substantial. They enable easy and frequent deployments, better testing by customers, smoother go-lives, and a general reduction in stress around project development and deployment cycles. The structured approach of SFDX ensures that deployments are reliable, repeatable, and less prone to errors.

To stay fact-based: SFDX deployments allow deploying multiple times a *week* in a few minutes per deployment. This allows ***very easy user testing***, and also allows finding *why* a specific issue cropped up. You can check the Examples section to see how and why this is useful.

> It is perfectly true that these deployments require more technical knowledge than third-party tools like Gearset or Changesets. It is our opinion that the tradeoff in productivity is worth the extra training and learning curve.

> One thing that is often overlooked - you can NOT do proper CI/CD without plugging the deployment to your project management. This means the entire project management MUST be thought around the deployment logic.

This training is split into the following chapters:

- **Chapter 1: The Why, When and By Whom** This chapter explores the fundamental considerations of Salesforce deployments within the context of consulting projects. It addresses:
  - **Why Deploy**: The importance and benefits of deploying Salesforce metadata throughout the project lifecycle, from the build phase to UAT to GoLive.
  - **When**: When in the project timeline should deployments be planned and executed to ensure smooth progress and mitigate risks. (Hint - it's often, but not in every org)
  - **By Whom**: Roles and responsibilities involved in the deployment process, such as consultants committing changes, architects reviewing commits and system elements, and release managers overseeing and executing deployments.

  Chapter 2: The What, How and How Frequently This chapter delves into the practical aspects of Salesforce deployments:
  - **What**: Overview of the deployment tools used, including GIT, SFDX, SGD, Bitbucket, and your trusty Command Line.

- **How**: Detailed workflows and methodologies for using these tools effectively, tailored to specific roles within the project team (consultants, architects, release managers).
- **How Frequently**: Recommendations on the frequency of deployments throughout the project timeline to maintain agility, minimize conflicts, and ensure continuous integration and delivery.

Chapter 3: An Example Project and Deployment Flow This chapter provides a hands-on example to illustrate a typical project scenario and the corresponding deployment processes:

- **Example Project**: Overview of a hypothetical Salesforce consulting project, including its scope and objectives.
- **Deployment Flow**: Step-by-step walkthrough of the deployment lifecycle, from initial planning and setup through to execution and validation.
- **Best Practices**: Highlighting best practices and potential challenges encountered during the deployment process.

Chapter 4: Configurations, Templates and Setup This chapter focuses on the essential configurations and setup required to streamline the deployment process:

- **Configurations**: Detailed guidance on configuring Salesforce environments for efficient deployment management.
- **Templates**: Templates and reusable patterns for standardizing deployments and ensuring consistency across projects.
- **Setup**: Practical tips and strategies for setting up deployment pipelines, integrating with version control systems, and automating deployment tasks.

These chapters collectively provide a comprehensive guide to mastering Salesforce deployments within a consulting company, covering both strategic considerations and practical implementation details.

Chapter 1: The Why, When and By Whom

This chapter explores the fundamental considerations of Salesforce deployments within the context of consulting projects. It addresses:

- **Why Deploy**: The importance and benefits of deploying Salesforce metadata throughout the project lifecycle, from the build phase to UAT to GoLive.
- **When**: When in the project timeline should deployments be planned and executed to ensure smooth progress and mitigate risks. (Hint - it's often, but not in every org)
- **By Whom**: Roles and responsibilities involved in the deployment process, such as consultants committing changes, architects reviewing commits and system elements, and release managers overseeing and executing deployments.

Why do I Deploy ?

In traditional software development, deployments often occur to migrate changes between environments for testing or production releases. However, in the context of Continuous Integration (CI) and Salesforce development, deployments are just synchronization checkpoints for the application, irrelevant of the organization.

Said differently, in CI/CD Deployments are just a way to push commits to the environments that require them.

> **CI deployments are frequent, automated, and tied closely to the development cycle.**
>
> **Deployments are never the focus in CI/CD, and what is important is instead the commits and the way that they tie into the project management - ideally into a ticket for each commit.**

> In software development, a **commit** is the action of saving changes to a version-controlled repository. It captures specific modifications to files, accompanied by a descriptive message. Commits are atomic, meaning changes are applied together as a single unit, ensuring version control, traceability of changes, and collaboration among team members.

Commits are part of using Git.
Git is a distributed version control system used to track changes in source code during software development. It is free and widely used, within Salesforce and elsewhere.

So if deployments are just here to sync commits...

Why do I commit ?

> As soon as a commit is useful, or whenever a day has ended.

Commits should pretty much be done "as soon as they are useful", which often means you have fulfilled **one** of the following conditions:

- you have finished working a ticket;
- you have finished configuring or coding a self-contained logic, business domain, or functional domain;
- you have finished correcting something that you want to be able to revert easily;
- you have finished a hotfix;
- you have finished a feature.

This will allow you to pull your changes from the org, commit your changes referencing the ticket number in the Commit Message, and then push to the repository.
This will allow others to work on the same repository without issues and to easily find and revert changes if required.

You should also commit to your local repository whenever the day ends - in any case you can squash those commits together when you merge back to Main, so trying to delay commits is generally a bad idea.

Take the Salesforce-built "Devops Center" for example.

They tie every commit to a Work Item and allow you to chose which elements from the metadata should be added to the commit. They then ask you to add a quick description and you're done.
This is the same logic we apply to tickets in the above description.

> If you're wondering "why not just use DevOps Center", the answer is generally "you definitely should if you can, but you sometimes can't because it is proprietary and it has limitations you can't work around".
> Also because if you learn how to use the CLI, you'll realise pretty fast that it goes WAY faster than DevOps Center.

To tie back to our introduction - this forces a division of work into Work Items, Tickets, or whatever other Agile-ism you use internally, and the project management level.

> **DevOps makes sense when you work iteratively, probably in sprints, and when the work to be delivered is well defined and packaged.**

This is because....

When do I Deploy ?

Pretty much all the time, but not **everywhere.**

In Salesforce CI/CD, the two main points of complexity in your existing pipeline are going to be:

- The first integration of a commit into the pipeline
- The merging of multiple commits, especially if you have the unfortunate situation where multiple people work in the same org.

The reasons for this are similar but different.

In the case of the first integration of a commit into the pipeline, most of the time, things should be completely fine. The problem is one that everyone in the Salesforce space knows very well. The Metadata API **sucks**. And sadly, SFDX... also isn't perfect.
So sometimes, you might do everything right, but the MDAPI will throw some file or some setting that while valid in output, is invalid in input. Meaning Salesforce happily gives you something you can't deploy.
If this happens, you will get an error when you first try to integrate your commit to an org. This is why some pre-merge checks ensure that the commit you did can be deployed back to the org.

In the case of merging multiple commits, the reasons is **also** that the Metadata API **sucks.** It will answer the same calls with metadata that is not ordered the same way within the same file, which will lead Git to think there's tons-o-changes... Except not really. This is mostly fine as long as you don't have to merge your work with someone else's where they worked on the same piece of metadata - if so, there is a non-zero chance that the automated merging will fail.

In both cases, the answer is "ask your senior how to solve this if the pipeline errors out". In both cases also, the pipeline should be setup to cover these cases and error out gracefully.

"***What does that have to do with when I deploy? Like didn't you get lost somewhere?***"

> The relation is simple - you should deploy pretty much ASAP to your remote repo, and merge frequently to the main work repository. You should also pull the remote work frequently to ensure you are in sync with others.
> Deploying to remote will run the integration checks to ensure things can be merged, and merging will allow others to see your work. Pulling the other's work will ensure you don't overwrite stuff.

> Deploying to QA or UAT should be something tied to the project management cycle and is not up to an individual contributor.
> For example, you can deploy to QA every sprint end, and deploy to UAT once EPICs are flagged as ready for UAT (a manual step).

Who Deploys ?

Different people across the lifecycle of the project.

On project setup, the DevOps engineer that sets up the pipeline should deploy and setup.
For standard work, you should deploy to your own repo, and the automated system should merge to common if all's good.
For end of sprints, the automated pipeline should deploy to QA.
For UAT, the Architect assigned to the project should run the required pipelines.

In most cases, the runs should be automatic, and key points should be covered by technical people.

Chapter 2: Software List

This chapter explores the actual tools we are using in our example, the basic understanding needed for each tool, and an explanation of why we're doing things this way.

In short, our example relies on:

- **Git**
  - A **Git** frontend if you are unused to Git - **gitkraken** is nice for Windows, or **sourcetree.**
  - There's a quite nice VSCode extension that handles Git properly.
- **Bitbucket**
- A good text editor (**VSCode** is fine, I prefer Sublime Text)
- The SF command line
  - **SFDMU**, a SF command line extension
  - **SGD**, a SF command line extension
  - **Code Analyzer**, a SF command line extension
- **JIRA**
- **A terminal emulator** (Cmdr is nice for windows, iTerm2 for MAC is fine)

You can completely use other tools if your project, your client, or your leadership want you do use other things.
The main reason we are using these in this example is that it relies on a tech stack that is very present with customers and widely used at a global level, while also leveraging reusable things as much as possible - technically speaking a lot of the configuration we do here is directly reusable in another pipeline provider, and the link to tickets is also something that can be integrated using another provider.

> **In short "use this, or something else if you know what you're doing".**

So What are we using

The CLI

The first entrypoint into the pipeline is going to be the **Salesforce Command Line.**You can download it here.
If you want a graphical user interface, you should set up VSCode, which you can do by following

this Trailhead. You can start using the CLI directly via the terminal if you already know what you're doing otherwise. If you're using VSCode, download Azul as well to avoid errors down the line.

> We'll be using the Salesforce CLI to:
> - login to organizations, and avoid that pesky MFA;
> - pull changes from an organization once our config is done;
> - rarely, push hotfixes to a UAT org.

For some roles, mainly architects and developers, we will also use it to:

- validate deploys to specific orgs in cases of hotfixes;
- setup SGD jobs in cases of commit-based deploys or destructive changes;
- setup SFDMU jobs for any data-based transfers.

What this actually does is allow you to interact with Salesforce. We will use it to get the configuration, security, and setting files that we will then deploy.

> This allows us not only to deploy, but also to have a backup of the configuration, and an easy way to edit it via text edition software.

The configuration needed is literally just the installation to start - we'll set up a full project later down the line.

GIT

You'll then need to download Git, as well as a GUI if you're not used to using it directly from the command line. Git is VERY powerful but also quite annoying to learn fully, which is why we will keep its usage simple in our case.

> We'll be using Git to:
> - version our work so we can easily go back to earlier configurations in case of issues;
> - document what we did when we modified something;
> - get the work that other people have done;
> - upload our work to the repositories for the project.

You'll need a bit more configuration once you're done installing - depending on the GUI you use (or if you're using the command line) the *how* depends on the exact software, but in short you'll need to configure git with your user name and your user email.

> Logging in to Bitbucket and getting your repository from there will come later - once you've given your username and email, and configured your UI, we will consider that you are done for now.

If you're a normal user, this is all you'll see of git.
If you're a Dev or an Architect, you'll also be using the Branches and Merges functions of Git - mostly through the Bitbucket interface (and as such, with Pull Requests instead of Merges).

Bitbucket

As said in intro, we're using bitbucket because we're using bitbucket. You can use Github, Gitlab, Gitea, whatever - but this guide is for bitbucket.

Bitbucket, much like Salesforce, is a cloud solution. It is part of the Atlassian cloud offering, which also hosts JIRA, which we'll be configuring as well. You'll need to authenticate to your workspace (maybe get your Administrator to get you logins), in the format https://bitbucket.org/myworkspace

You will see that Bitbucket is a Git Server that contains Git Repositories.
In short, it is the central place where we'll host the different project repositories that we are going to use.
Built on top of the Git server are also subordinate functions such as Pull Requests, Deployments, Pipelines - which we're all going to use.

Seeing as we want this to be connected with our Atlassian cloud, we'll also ask you to go to https://bitbucket.org/account/settings/app-passwords/ which allows you to create application passwords, and to create one for Git.

In detail:

- **Repositories:** Developers store their Salesforce metadata and code in Bitbucket repositories. Each repository can represent a project or a component of a larger Salesforce application.
- **Branching:** Developers create branches for new features, bug fixes, or enhancements. This allows multiple developers to work on different parts of the codebase simultaneously without interfering with each other.
- **Pull Requests:** When a feature or bug fix is complete, a pull request is created. Other team members review the changes before they are merged into the main branch, ensuring code quality and consistency.
- **Commits:** Developers commit their changes to Bitbucket, providing a detailed commit message. These messages often include references to JIRA ticket numbers (e.g., "Fixed bug in login flow [JIRA-123]").
- **JIRA:** When a commit message includes a JIRA ticket number, JIRA can automatically update the status of the ticket, link the commit to the ticket, and provide traceability from issue identification to resolution.

- **Pipelines:** Bitbucket Pipelines can be configured to automatically build, test, and deploy Salesforce code changes. This ensures that changes are validated before being merged and deployed to production. It does so using **Deployments** - which in bitbucket means "the installation of code on a remote server", in our case Salesforce.

Extra Stuff

CLI Extensions

SGD

SGD, or Salesforce-Git-Delta is a command line plugin that allows the CLI to automatically generate a package.xml and a destructivechanges.xml based on the difference between two commits.
It allows you to do in Git what the CLI does alone using Source Tracking.
Why is it useful then ? Because Source Tracking is sometimes buggy, and also because in this case we're using Bitbucket, so it makes generating these deployment files independent from our machines.
SGD is very useful for inter-org deployment, which should technically be quite rare.

SFDMU

SFDMU, or the Salesforce Data Move Utility, is another command line plugin which is dataloader on steroids for when you want to migrate data between orgs or back stuff up to CSVs.
We use this because it allows migrating test data or config data (that last one should be VERY rare what with the presence of CMTD now) very easily including if you have hierarchies (Contacts of Accounts, etc).

Code Analyzer

AzulJDK

Basically just Java, but free. We don't use the old Java runtime because licensing is now extremely expensive.

A terminal emulator

If you don't spend a lot of time in the Terminal, you might not see that terminals aren't all equal.
A nice terminal emulator gives you things like copy/paste, better UX in general.
It's just quality of life.

A text editor

You should use VSCode unless you really want to do everything in separate apps.
If you're an expert you can use whatever floats your boat.

Chapter 3: Basic Machine Setup

1 - Install Local Software

If you are admin on your machine, download Visual Studio Code from this link. Otherwise, use whatever your IT has to install software, whether it be Software Center, opening a ticket, or anything else of that ilk.

As long as you're doing that, you can also install a JDK like AZUL, as well as Git, and a nice terminal emulator.

Also remember to install the Salesforce CLI.

> These elements are all useful down the line, and doing all the setup at once avoids later issues.

2 - Configure the CLI

Opening your beautiful terminal emulator, run

```
sf update
```

You should see `@salesforce/cli: Updating CLI` run for a bit.

> If you see an error saying `sf` is not a command or program, something went wrong during the installation in step 1. Contact your IT (or check the installation page of the CLI if you're Admin or not in an enterprise context).

Once that's done, run

```
echo y | sf plugins install sfdmu sfdx-git-delta code-analyze
```

> Because sgd is not signed, you will get a warning saying that "This plugin is not digitally signed and its authenticity cannot be verified". This is expected, and you will have to answer `y` (yes) to proceed with the installation.

Once you've done that, run:

`git config --global user.name "FirstName LastName"` replacing Firstname and Lastname with your own.

`git config --global user.email "email@server.tld"` replacing the email with yours

If you're running Windows - `git config --global core.autocrlf true`

If you're running Mac or Linux - `git config --global core.autocrlf input`

> The above commands tell git who you are, and how to handle line endings.
> All of this setup has to be done once, and you will probably never touch it again.

Finally, run

`java --version`

> If you don't see an error, and you see something like `openjdk 21.0.3 2024-04-16 LT` then you installed Zulu properly and you're fine.

3 - Configure VSCode

Open up VSCode.

Go to the Extensions in the side panel (it looks like three squares)  and search for "Salesforce", then install

- Salesforce Extensions Pack
- Salesforce Extensions Pack (Expanded)
- Salesforce Package.xml Generator for VS Code
- Salesforce CLI Command Builder
- Salesforce XML Formatter

Then search for Atlassian and install "Jira and Bitbucket (Atlassian Labs)".

Finally, search for and install "GitLens - Git supercharged".

Then go to **Preferences > Settings > Salesforcedx-vscode-core: Detect Conflicts At Sync** and check this checkbox.

Once all this is done, I recommend you go to the side panel, click on Source Control, and drag-and-drop both the Commit element and the topmost element to the right of the editor.

> All this setup allows you to have more visual functions and shortcuts. If you fail to install some elements, it cannot be guaranteed that you will have all the elements you are supposed to.

> This concludes basic machine setup.
> All of this should not have to be done again on an already configured machine.

Chapter 4 - Base Project Setup

This chapter explores how to set up your project management and version control integration, ensuring proper tracking from requirement to deployment.

Initial Project Creation

SFDX Project Setup

Create Base Project

```
sf project generate
    --name "your-project-name"
    --template standard
    --namespace "your_namespace"  # if applicable
    --default-package-dir force-app
```

Required Project Structure

```
your-project-name/
├── config/
│   └── project-scratch-def.json
├── force-app/
│   └── main/
│       └── default/
├── scripts/
│   ├── apex/
│   └── soql/
├── .forceignore
├── .gitignore
```

```
├── package.json
└── sfdx-project.json
```

Configuration Files Setup

`.forceignore` Essential Entries

```
# Standard Salesforce ignore patterns
**/.eslintrc.json
**/.prettierrc
**/.prettierignore
**/.sfdx
**/.sf
**/.vscode
**/jsconfig.json

# Package directories
**/force-app/main/default/profiles
**/force-app/main/default/settings
```

`.gitignore` Essential Entries

```
# Salesforce cache
.sf/
.sfdx/
.localdevserver/

# VS Code IDE
.vscode/

# System files
.DS_Store
*.log
```

Bitbucket Repository Integration

Initial Repository Setup

In Bitbucket:

```
- Create new repository
- Repository name: your-project-name
- Access level: Private
- Include README: Yes
- Include .gitignore: No (we'll use our own)
```

Linking Local Project to Remote Repository

Initialize Git Repository

```
cd your-project-name
git init
git add .
git commit -m "Initial project setup"
```

Link to Bitbucket

```
git remote add origin https://bitbucket.org/your-workspace/your-project-name.git
git push -u origin main
```

Branch Protection Rules

Configure in Bitbucket Repository Settings:

YAML

```
Branch Permissions:
  main:
```

```
    - Require pull request approvals
    - Minimum approvers: 2
    - Block force pushes
  develop:
    - Require pull request approvals
    - Minimum approvers: 1
    - Block force pushes
```

Project Configuration Files

`sfdx-project.json` Configuration

JSON

```
{
    "packageDirectories": [
        {
            "path": "force-app",
            "default": true,
            "package": "your-project-name",
            "versionName": "Version 1.0",
            "versionNumber": "1.0.0.NEXT"
        }
    ],
    "namespace": "",
    "sourceApiVersion": "60.0"
}
```

`project-scratch-def.json` Base Configuration

JSON

```
{
    "orgName": "Your Project Name",
    "edition": "Enterprise",
    "features": ["EnableSetPasswordInApi"],
    "settings": {
        "lightningExperienceSettings": {
            "enableS1DesktopEnabled": true
        },
        "securitySettings": {
```

```
            "passwordPolicies": {
                "enableSetPasswordInApi": true
            }
        }
    }
}
```

Post-Setup Verification

Run these commands to verify setup:

Bash

```
# Verify SFDX project
sf project verify

# Verify Git setup
git remote -v

# Verify Bitbucket connection
git fetch origin

# Verify branch protection
git push origin main --dry-run
```

JIRA Configuration

- Create a new project
- Configure the following required elements:
  - Epic issue type
  - Story issue type
  - Bug issue type
  - Task issue type
  - Custom fields for Salesforce metadata tracking

Required JIRA Workflow States

Text Only

```
Backlog -> In Progress -> In Review -> Ready for Deploy -> Done
```

Bitbucket Integration

- Link your JIRA project to Bitbucket repository
- Configure repository access rights
- Setup branch policies:
  - `main` - protected, requires PR
- `develop` - protected, requires PR
- `feature/*` - development branches
- `hotfix/*` - emergency fixes

Work Segmentation

Story Creation Rules

Stories should be:
- Independent (can be deployed alone)
- Small enough to be completed in 1-3 days
- Tagged with proper metadata types
- Linked to an Epic

Required Story Fields

- Epic Link
- Acceptance Criteria
- Metadata Types
- Development Notes
- Test Cases

Integration Setup

JIRA to Bitbucket Connection

1. In JIRA:
   - Navigate to Project Settings
   - Enable "Development" integration
   - Link to Bitbucket repository

2. In Bitbucket:
    - Configure branch policies
    - Setup automatic JIRA issue transitions
    - Enable smart commits

Commit Message Format

Text Only

```
[PROJ-123] Brief description


- Detailed changes
- Impact on existing functionality
- Related configuration
```

Pipeline Configuration

Get the bitbucket-pipelines.yml file

Integrate it and set up the variables

Automation Rules

JIRA Automation

- Create branch from ticket
- Update ticket status on commit
- Link PR to ticket
- Transition on successful deployment

Bitbucket Pipelines

- Trigger on PR creation
- Run validation suite
- Deploy to appropriate environment
- Update JIRA ticket status

Security and Access

Required Team Roles

- Project Admin (JIRA + Bitbucket)
- Development Team (restricted repository access)
- Release Manager (deployment rights)
- QA Team (environment access)

Access Matrix

Text Only

```
Role          | JIRA |  Bitbucket |  Salesforce
Project Admin |  Admin|  Admin     |  System Admin
Developer     |  Write|  Write     |  Developer
QA           |  Write|  Read      |  Read-only
```

Remember that this setup needs to be done only once per project, but maintaining the discipline of following these structures is crucial for successful CI/CD implementation.

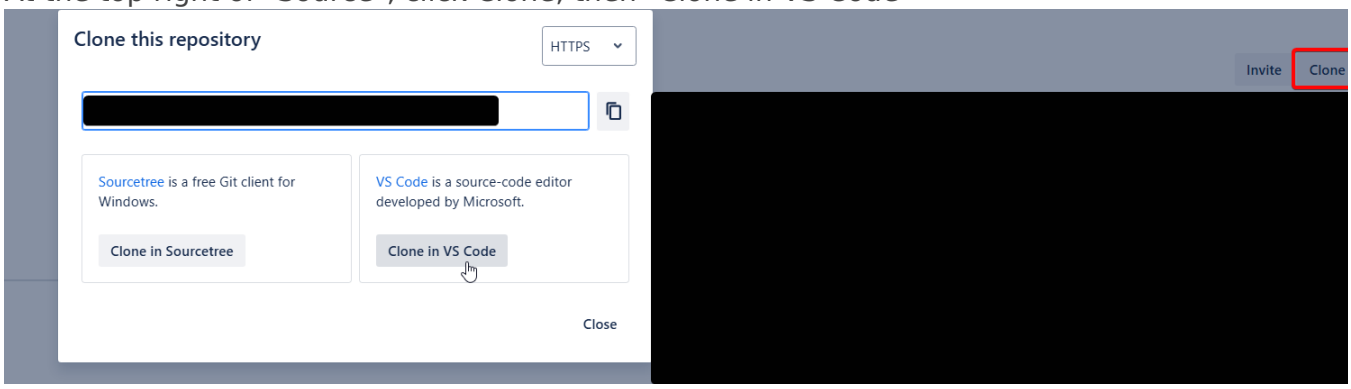
The key to success is ensuring that:
1. Every piece of work has a ticket
2. Every commit links to a ticket
3. Every deployment is traceable
4. All changes are reviewable

This structured approach ensures that your project management directly ties into your deployment pipeline, making it easier to track changes and maintain quality throughout the development lifecycle.

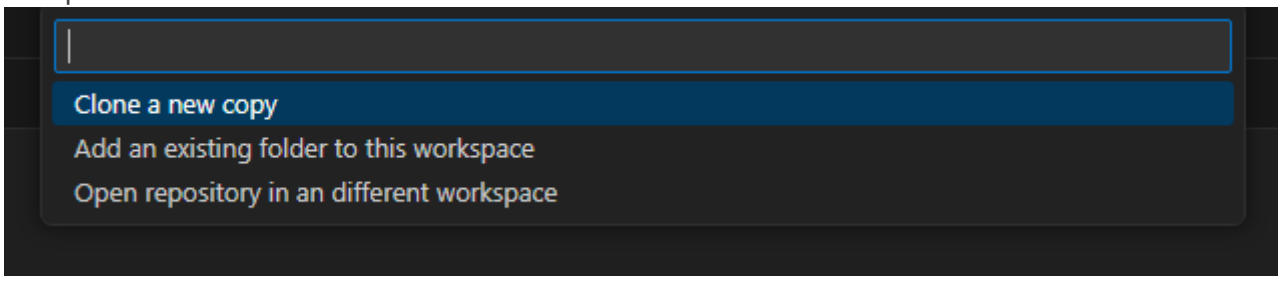
Chapter 5 - Using an Existing Project

Repository

1. Go to Bitbucket
2. Go the repository for the project.
3. At the top right of "Source", click Clone, then "Clone in VS Code"



4. Allow VSCode to open the link and install the extension if you are requested to do so. Also trust the publisher "Atlassian".
    1. If this is the first time, click on "Login to JIRA", follow the steps (including "back to vs code"), then also login to Bitbucket using the same screen.
5. If during the Clone, Git requests a login, you might need an App Password. You can create one by going to the cog on the top right, going to Personal Bitbucket Settings, and App passwords.
6. VScode will ask what to do - select "clone a new copy", and store it somewhere on your computer.



7. Log in to the Org that you will be working against. Presumably a scratch org, but if you're still using persistent Orgs, this means login to Production, and Dev orgs as needed, using
8. Ensure your workspace has Source Source Tracking, Atlassian, and Salesforce icons.

Salesforce

1. Go to your Production, then navigate to Setup > Dev Hub. ensure `Enable Dev Hub` and `Enable Source Tracking in Developer and Developer Pro Sandboxes` are both checked.

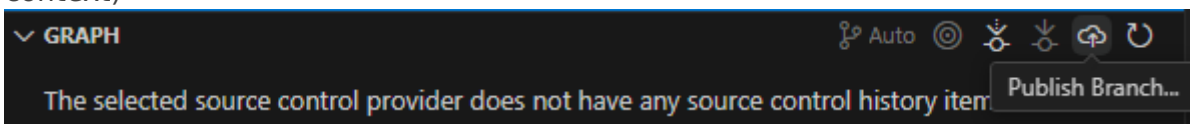
> These settings allow both the CLI and Change Sets to function with Source Tracking. This fully replaces the usual "compare the source and the target organizations

VSCode

1. Open VSCode and open your Repository Folder using VSCode.
2. If necessary, run `sf org login --alias MyOrg` with MyOrg being the org type (examples: `Client-Prod`, `Client-Dev`, `Client-ScratchFeatureName`)
3. Start working in Salesforce, then go the Terminal and run `sf project retrieve start --target-org MyOrg --ignore-conflicts --wait 20` where MyOrg is the organization you have logged in to which contains what you want. This assumes Source tracking is on.

> This works best if you are the only one working in an Org. If you are not, you might want to start using Scratch orgs. Otherwise, you will see all the changes done by everyone.

1. Go the Terminal and run `sf project deploy start --target-org MyOrg --ignore-conflicts --wait 20` where MyOrg is the organization you have logged in to which must receive what you got. This assumes Source tracking is on. If Source Tracking is not on, or you want to do a full send, you can run `sf project deploy start --target-org MyOrg --source-dir force-app --ignore-conflicts --wait 20`
2. Go to Source tab in VSCode, then add a commit message, Stage the changes required, and Commit.
3. Push your changes (the cloud with an upwards arrow - name may change based on context)

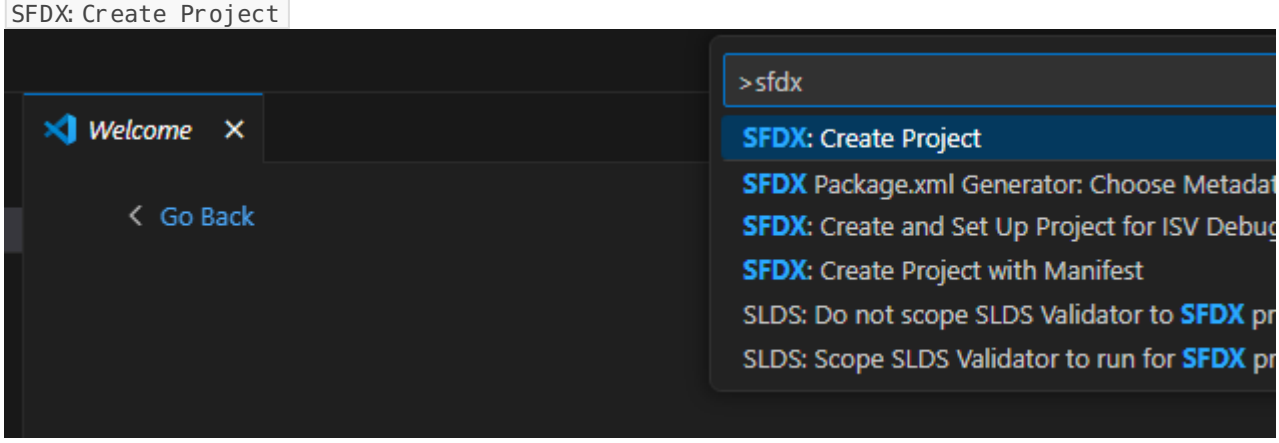


Chapter 6 - Creating a New Project

Setting up the Project

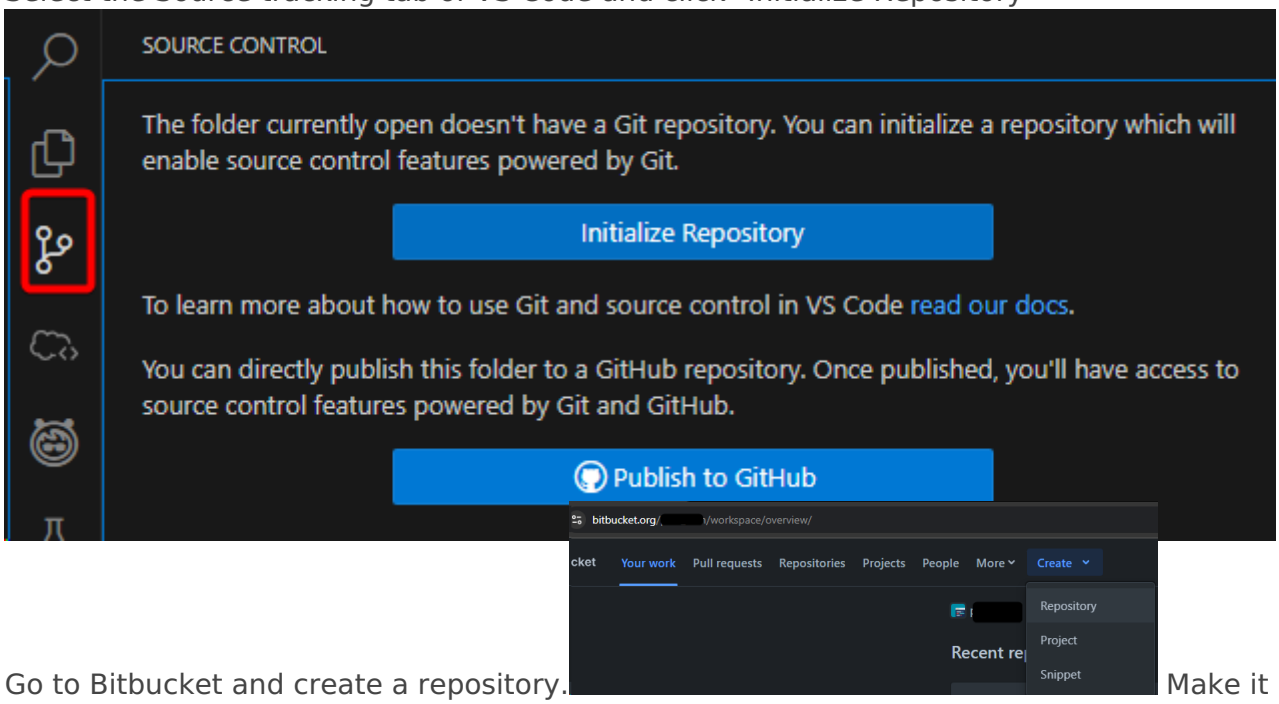
SF Cli Project

1. Open VSCode and navigate to the folder where you want to create the Project - presumably `somefolder/sfdx_projects` or equivalent
2. Use CTRL+SHIFT+P to open the VSCode Command Panel and type SFDX, then select `SFDX: Create Project`



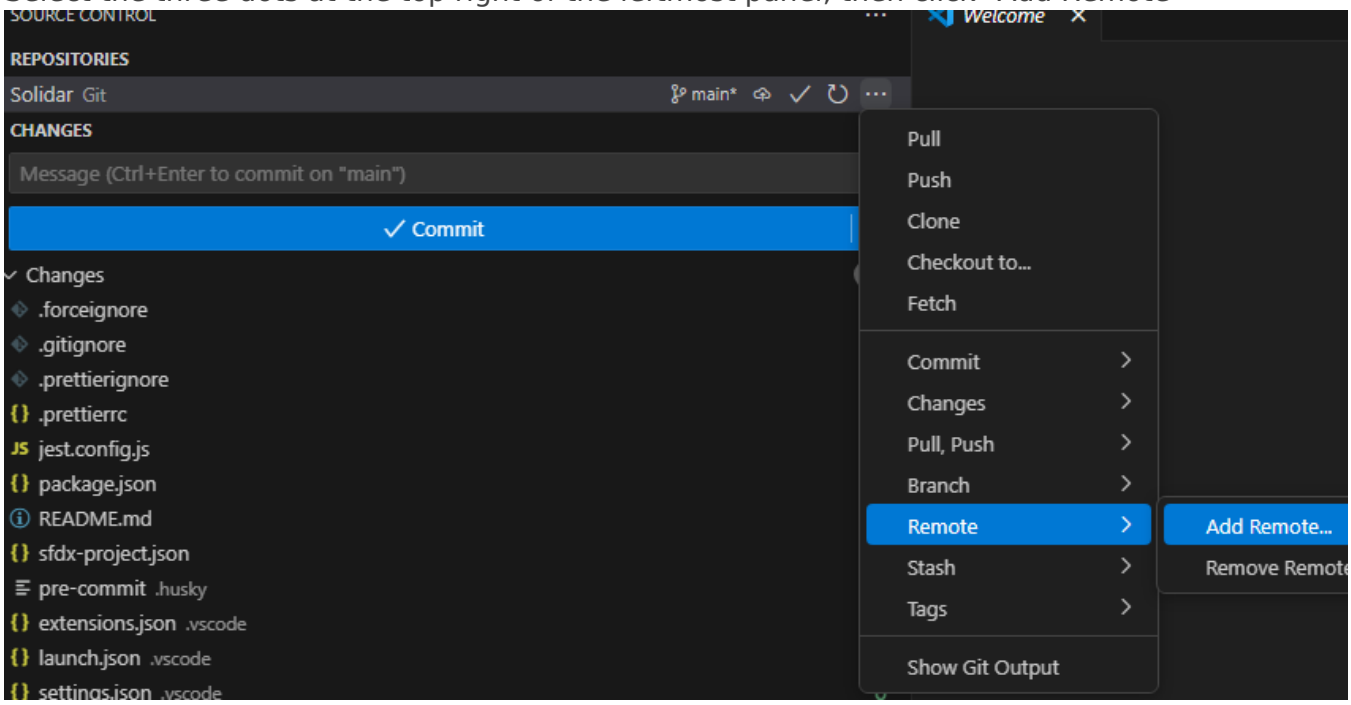
3. Select `empty` then type the name of your project and type Enter.
4. This will create a new SF Project. VS Code will ask if you want to open the folder - say yes.
5. You should now have the VScode Project open.

6. Select the Source tracking tab of VS Code and click "Initialize Repository"



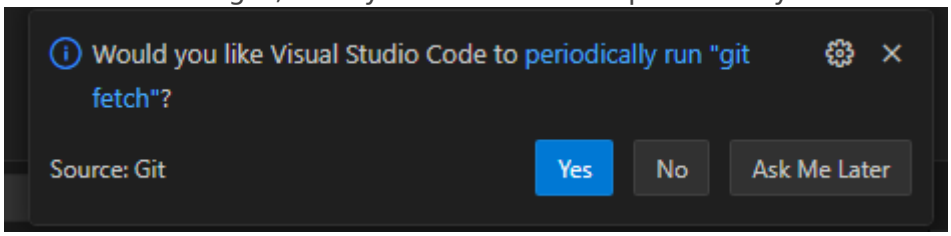
7. Go to Bitbucket and create a repository. Make it "empty" and refuse to add a GitIgnore.

8. Copy the URL given by Bitbucket in the format `https://MYNAME@bitbucket.org/MYWORSPACE/MYREPO.git` which should be displayed in the screen.
9. Select the three dots at the top right of the leftmost panel, then click "Add Remote"



10. Paste the URL, select "Add Remote from URL". When asked for the remote name, type `origin` then press enter.

11. At the bottom right, click yes on "Run Fetch periodically".



12. run `git push --set-upstream origin main` in the VSCode Terminal to sync your local and remote branches.

Fetch your Salesforce Metadata

1. Login to your Salesforce org. This can be done using CTRL+SHIFT+P `Authorize an Org`, or running `sf org login web --alias MyOrg` in the terminal where MyOrg is the name you want to give to the org.
2. Run `sf project retrieve start --target-org MyOrg --ignore-conflicts --wait 20`
3. Commit the state of the project and push

Continue working

Useful Command reference

Base

```
sf org login web --alias MyOrg
```

> Login

```
git add . && git commit -m "My Commit Message" && git push
```

> quick and dirty commit everything from local

```
git remote update origin --prune
```

> get all Branches from remote and remove the deleted ones from your local

Deployments

```
sf project retrieve start --target-org MyOrg --ignore-conflicts --wait 20
```

Retrieve changes from Org. Assumes Source Tracking is on.

```
sf project retrieve start --target-org MyOrg --ignore-conflicts --wait 20  -x manifest/package.xml
```

> Get the metadata listed in the package from the Org

```
sf project retrieve start --target-org MyOrg --ignore-conflicts --wait 20  -m InstalledPackage:FSL
```

> Retrieve specific metadata from the org. Based on the Metadata Types from SF https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_types_list.htm

```
sf project deploy start --target-org MyOrg --ignore-conflicts --wait 20
```

> Deploy the source. Does destructive changes and assumes source tracking

```
sf project deploy start --target-org MyOrg --source-dir force-app --ignore-conflicts --wait 20
```

> Deploy the metadata contained in the force-app folder. Does not do destructive changes.

```
sf project deploy start --target-org MyOrg --manifest package/package.xml --pre-destructive-changes destructiveChanges/destructiveChanges.xml --ignore-conflicts --ignore-warnings --wait 20 --dry-run
```

> Same as previous but validates only

Using SGD

```
sf sgd:source:delta --to HEAD --from HEAD~1 --output-dir . -i .sgdignore
```

> sgd - do delta from last commit to current commit, generate the package.xml and destructivechanges in the current directory, and ignore directories listed in .sgdignore

```
sf project deploy start --target-org MyOrg --manifest package/package.xml --pre-destructive-changes destructiveChanges/destructiveChanges.xml --ignore-conflicts --ignore-warnings --wait 20 --dry-run
```

> sgd - do a Salesforce Project deploy based on the generated manifest from SGD, and do destructive changes before running anything else based on the destructivechanges.xml