

# Salesforce Packages : Pattern Strategy

## 1 INTRODUCTION

---

This document provides a suggested pattern for creating Reusable package components for Salesforce. Even though the document addresses Unlocked Packages, the organization and strategy will not vary much. This document assumes the user has a basic level understanding of the Salesforce DX.

This is a living document as Wells Fargo will discover a process that fits their organization values and viewpoints. A packaging strategy can, and will vary, over time and one should expect to review the process semi-annually or annually to ensure proper compliance and efficiencies.

## 2 GOAL ALIGNMENT

---

The document outlines a package strategy aligned to the following Wells Fargo's stated goals:

- Scalability – support federated development model
- Teams release independent from each other
- Increase developer efficiency (and focus)

## 3 OUT OF SCOPE

---

- Areas such as Source Control Structure, Jenkins, Release Management Strategy and CI/CD are not covered.

## 4 PACKAGES (UNLOCKED)

---

Salesforce DX Packages provide an ability to create metadata compartments for development and deployment into modular packages; allowing for an independent development and deployment experience. However, in order to maximize your DX experience, it requires a paradigm shift from common Salesforce *Org* development.

This document outlines a change in development. It requires careful planning and an assortment of development standards to avoid creation of one large package with no dependencies.

### 4.1 KEY BENEFITS

- Follows best practices regarding the software development life cycle. It's compatible with the new features of Salesforce DX: projects, source-driven development commands, and scratch orgs were built specifically with packaging in mind.

- Encapsulates all the changes you are tracking between life cycle stages in a versioned artefact (i.e. Git).
- Promotes iterative and modular development.
- Supports interdependencies among unlocked packages. A single unlocked package can depend on multiple unlocked packages.
- Supports continuous integration and continuous delivery because the packaging CLI commands enable each step in the deployment pipeline to be fully automated.
- Provides an improved audit history, so you can more easily track and understand the changes made to a production org.

## 4.2 GENERAL GUIDELINES FOR PACKAGES

### 4.2.1 Base package:

- Common interfaces
- Responsible for standard Salesforce *SObjects*
- Focused on holding all the object-level metadata (and custom metadata types)

### 4.2.2 Dependencies

- By default, when referencing a certain Standard Object, field, or component type, you will generate a prerequisite dependency on your package

### 4.2.3 Breaking Down Metadata Across Objects

- A group of related code and customization
- Can be tested independently from other components in your org
- Able to be released independently
- Source components can only live in one project at a time

### 4.2.4 Things that don't work

- [Shared metadata](#) (i.e. Two packages sharing the same metadata)
- [Shared Objects](#) (i.e. Two packages sharing the same custom object)

## 5 THREE MODELS FOR UNTANGLING YOUR METADATA

---

In real-life scenarios, there are various combinations of the strategies below. The document tweaks them to best suit the business needs.

Model	Description
App-based	Identify metadata that represents an app. This approach is similar to creating a package for an Application (e.g. <a href="#">Dreamhouse</a> ) with the exception that the metadata already exists in your org.

Model	Description
Customizations-based	Organize unpackaged metadata for customizations and functionality changes in your production org, such as customizations to Sales Cloud, Service Cloud, or an Wells Fargo (WF) AppExchange app.
Shared library	When interdependencies exist, use a common Salesforce DX package to organize a set of Apex classes or commonly used custom objects. Other packages that you construct can depend on this common package.

## 5.1 PACKAGE SLICING

Below are two perspectives on package slicing<sup>1</sup>.

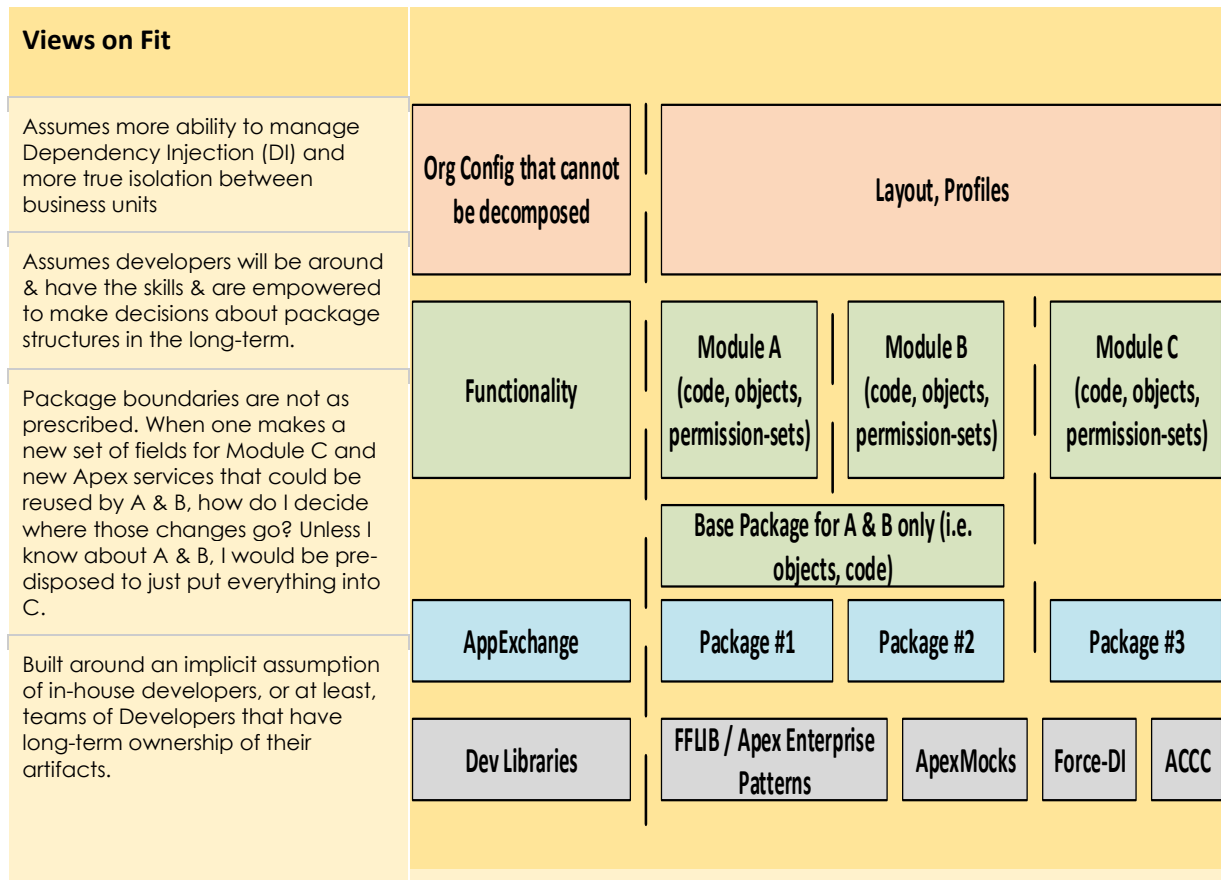
### 5.1.1 Vertical Slicing

Flexible, more advanced code required, packages are developer-driven.

Pros	Cons
Packages are more self-contained and managed.	Only possible when there is a clear set of objects that can be associated with a module/application.
Dependencies on AppExchange packages are more isolated, dev environments and scratch orgs can be more straightforward to set up.	Requires a cap-stone package (or just left in the org) for all the things you cannot decompose at present (i.e. Metadata Dependency)
Supports the option to go horizontal in places, if needed	Governance is more complex (due to distribution of modules)

---

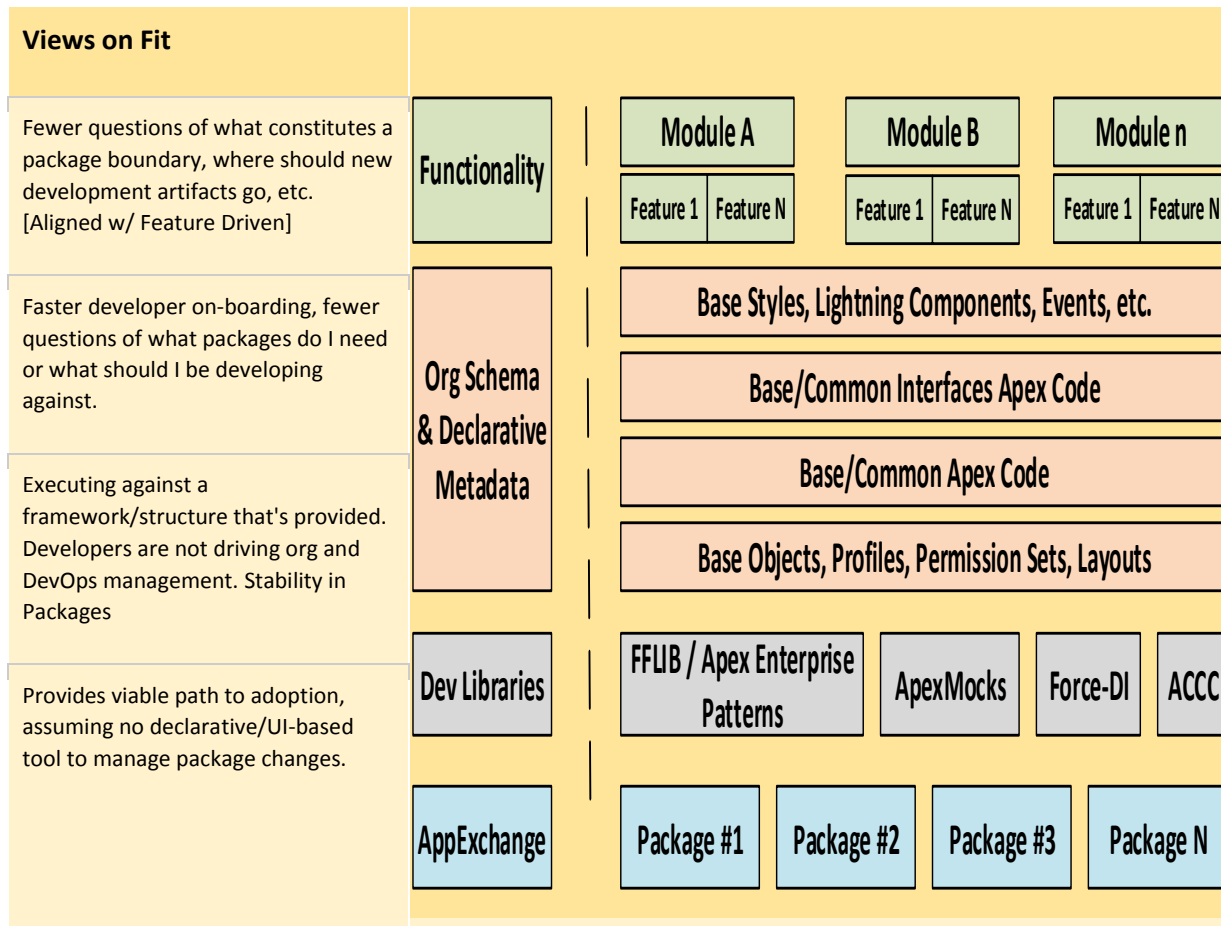
<sup>1</sup> *Unlocked Packages [slicing] (internal Salesforce)* – Abhishek Sinha



### 5.1.2 Horizontal Slicing

More prescribed, less advanced code required, easier for declarative.

Pros	Cons
Shared packages for shared metadata, code can help org governance	Large packages that change frequently can add overhead
Less prone restrictions relating to inability to untangle objects, etc.	
Allows option to use vertical slicing where needed	



## 5.2 PROPOSED PACKAGE STRUCTURE

Some general guidelines around package structure are provided below.

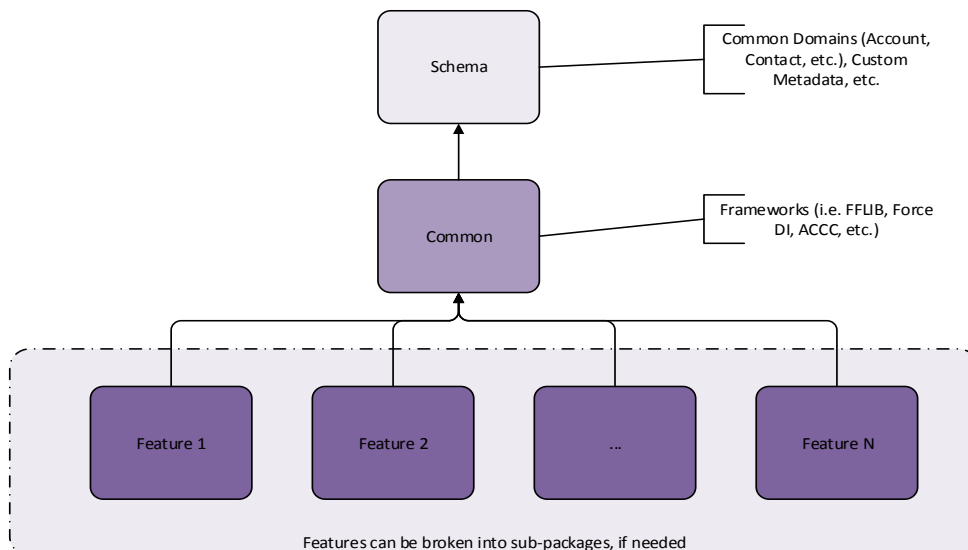
**Guideline:** Break Happy Soup Org into small packages following the guideline of least likely to change to most likely to change. For example, defining a custom object, the structure is not likely to change often. Isolating the custom object into a **Schema** package will help insulate a need to checkout often when other (dependent) aspects change.

- **Schema:** Package contains the foundational elements of the Org centered around SOjects, Custom Metadata, Validation rules, etc.
- **Core:** Package contains any architectural framework classes and effectively forms the base package; all other packages are dependent on this. Suggestions are:
  - Put core functionality in a core package. For example, all packages should utilize the [Apex Enterprise Framework/Pattern](#) as this provides a **separation of concerns (SOC)** in Apex development (also provides a path for reuse):

- [Selector \(Data Layer\)](#)
- [Domain \(Common Layer\)](#)
- [Service \(Business Layer\)](#)

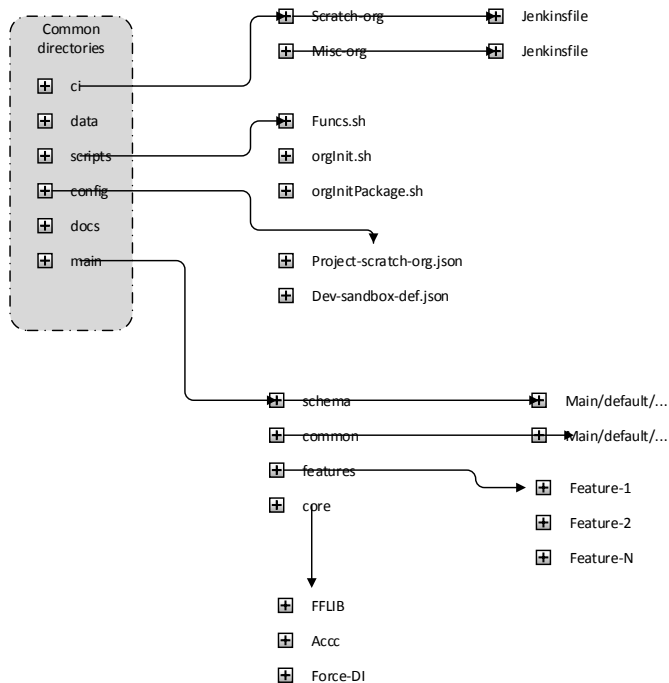
- Core components will mostly likely have dependencies (e.g. Logging, Trigger Framework, etc.)

- **Common:** Contains common domain (e.g. Account, Contact, Case, Activities, Custom Objects, etc.) related items (e.g. Helpers, Trigger handlers, etc.) . These domains may be further extended in the *feature* packages; but it is important to maintain a centralized common base to avoid unwanted duplication (e.g. multiple triggers on the same object).
- **Feature Packages:** Most likely to be aligned with project initiatives or specific domains (i.e. not 1:1 with JIRA features): composition should be determined on a project by project basis as part of Plan & Architect phase.



- **Scripts** – This directory contains common scripts to push/deploy to Scratch Orgs and Sandboxes (e.g. Dev-Orgs).
- **Ci** – This directory contains commonly used Jenkins files for Scratch Orgs and non-Scratch Orgs.
- **Data** – This directory holds data to seed the Sandbox or Scratch Org.

A project configuration would be as follows :



Please note, the structure above is *ALL* encompassing. For example, most packages will not include directories such as **FFLIB**, **Accc**, etc. as these would be package as dependencies in the *sfdx-project.json*. In addition, some projects may not have **data** or **docs** directory.

### 5.3 FEATURE PACKING PRINCIPLES

A package comprises a group of related code and customizations

- A package can be tested independently from other components in the org
- A package should be able to be released independently from other components (excluding dependencies on Core / Common)
- Core package is reserved for common used architecture frameworks and not configuration metadata (e.g. objects etc.)
- Metadata components within a package can only exist in one package at a time. For example the same field should not exists in multiple packages.
- Do not pollute the Core / Common packages
- Test class coverage should be managed per package (as well as overall) and should not be lower than entry criteria

## 5.4 CONSIDERATIONS / LIMITATIONS

- Salesforce DX and Unlocked Packages are relatively new feature for enterprise application development - there are still some limits and is still evolving. For example, you will need to ensure package components are supported via the Metadata Dependency Coverage [here](#).
- The packaging API that Salesforce DX is built on top of does not yet support all the objects that the metadata API supports (which in turn does not have full coverage). As such, some manual steps will still be required
- Profile migration is not supported by the packaging API and needs to be addressed separately. To some extent this can be mitigated through more extensive use of permission sets; however some preferences (e.g. assignment of record types) can only be set at profile level

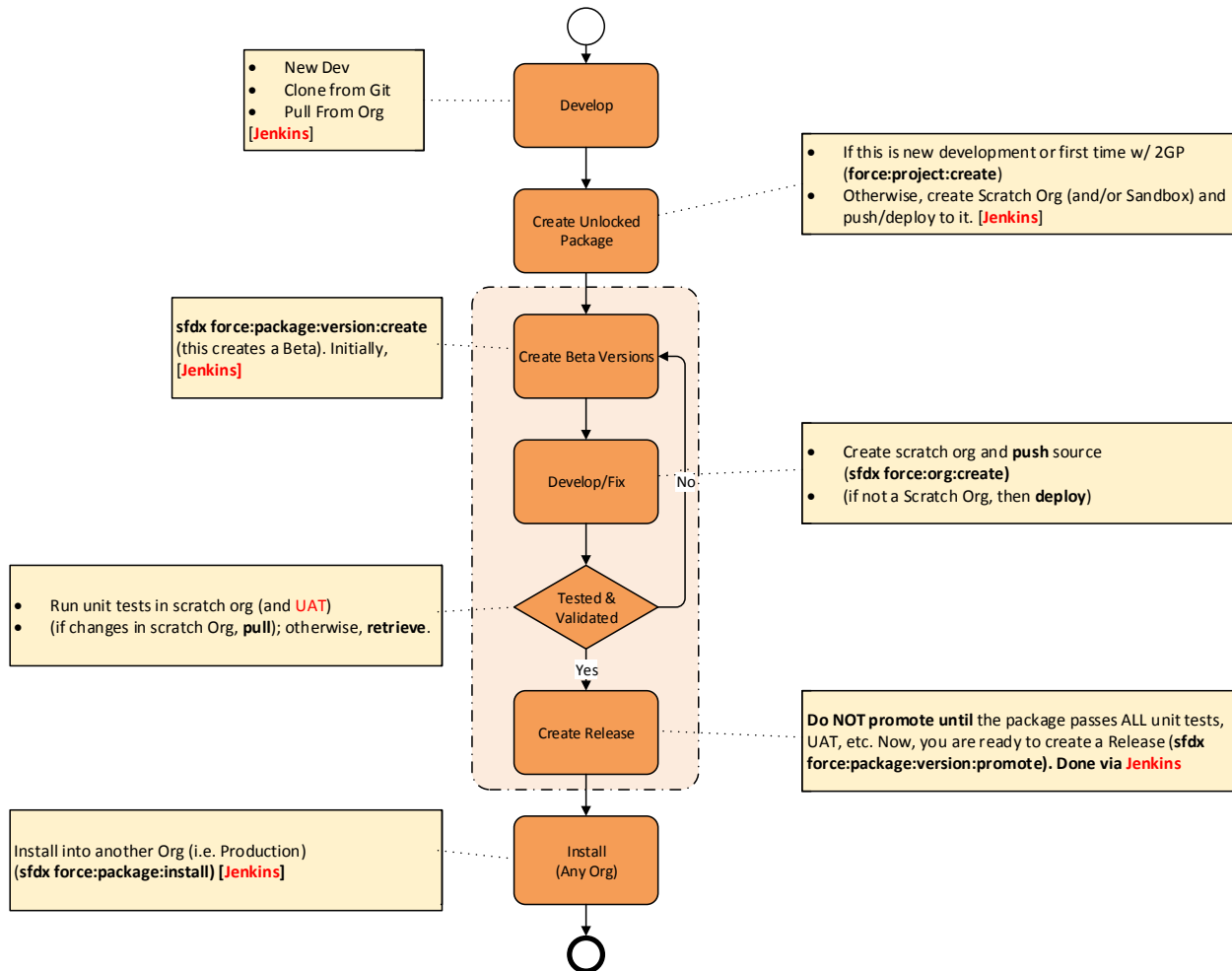
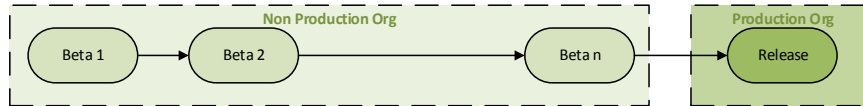
## 6 PACKAGE DEVELOPMENT PROCESS

---

Developers (and DevOps) working with Packages (Unlocked or Managed) go through distinct processes. DevOps is responsible for promoting and deploying into upstream Orgs (e.g. UAT, SIT, STAGE, PROD).

The diagram below outlines a high-level of the process. Items marked with **[Jenkins]** reflect actions by the DevOps team.





## 7 BREAKING THE HAPPY SOUP

When tackling an Org's [Happy Soup](#) there are a number of ways to address this process. There are combination of tools and solution discussed below to break the Happy soup into packages. One of the most important things to keep in mind is having a firm knowledge of the components (or the 1<sup>st</sup> generation package) you wish to moving into 2GP. Without this knowledge, the process will take longer, be more error prone, and not reflect a proper distillation into packages.

**Guideline:** Breaking up a 1GP package, or the Happy Soup, into one large package will limit the number of teams making changes. Smaller, more distinct packages, allows teams to update without collision (merge-conflicts) with other teams.

## 7.1 CREATE A PACKAGE WITHIN THE ORG

Salesforce Org allows one to create a package (Setup→Package Manager). This allows one to begin the process of breaking the Happy Soup as well as adding dependencies. You can then download the package. You can then remove and add components as necessary.

## 7.2 METADATA COMPONENT DEPENDENCY API

The API allows one to understand dependencies such as Apex Classes, Static Resources, Visualforce pages, etc. [Andy Fawcett](#)<sup>2</sup> provided a host of SFDX and SOQL commands to help understand these dependencies. In addition, Daniel Ballinger created a plugin ([FIT](#)<sup>3</sup>) to further analyze your Happy Soup.

## 7.3 APEX UML

A UML diagram can also help breakdown dependencies (inheritance/use) in a heavily used APEX Org. Some of the tools may not work within your Org but are listed here.

- [Plant UML](#)
- [APEX UML](#) (note, have had better luck with this tool; though, still a but clunky)

# 8 PACKAGE STRATEGY ALIGNMENT WITH GOALS

---

The table below provides the benefits of the package strategy aligned to the goals.

Category	Scale Ability	Release Independently	Increase Efficiency
Modular Development	X	X	X
Separation of Concerns		X	X
Reusable Components	X		X
Source-driven development		X	X
Automation	X		X

The table above outline how a category (e.g. Modular Development) benefits all three Goals. The sections below cover how the Goal is satisfied and associated KPI.

## 8.1 SCALABILITY

- *Modular development* provides a focus work-effort. How? A package is a smaller component, i.e. Account Selector, one may triage. The smaller component may be used in other packages. Thus, performance improvement or bug fix benefits not only the associate application but related applications.
- *Reusable Components*, if properly packaged, can benefit other applications. For example, the Logging Framework provides a common reusable package (i.e. logging, exception handling, etc.)

---

<sup>2</sup> Not all dependencies can be reported at this time

<sup>3</sup> Have not used this tool (yet)

that is written once but reused everywhere. One no longer writes the SAME code in multiple applications (avoid costly maintenance).

- *Automation*, allows the developer to test quickly and often while repeating the deployment process via Scratch Orgs.

#### 8.1.1 Expected Success Factors / KPIs

- Quicker delivery
- Less merge-conflicts
- Less exceptions; reduce maintenance cost
- Increased Business delivery

### 8.2 RELEASE INDEPENDENTLY

- *Modular development* provides a focus work-effort. How? A package is a smaller component, i.e. Account Selector, one may triage. The smaller component may be used in other packages. Thus, performance improvement or bug fix benefits not only the associate application but related applications.
- *Separation of Concerns*, if properly packaged, reduces the collision (cross layered). For example, dividing an application into Business, Data and Presentation layers three different teams can work independently once you define the common interface. The common interface (Core) provides an agreed upon contract whereby, each team can work, without much, if any, collision [i.e., merge-conflicts].
- *Source-driven development*, provides a greater agility to test out features with confidence (as the dependent packages are stable and have already been tested)

#### 8.2.1 Expected Success Factors / KPIs

- Quicker delivery
- Less merge-conflicts
- Increased Business delivery

### 8.3 INCREASE EFFICIENCY

The previous two goals combined, above, already outline Modular development, Separation of Concerns, Reusable Components and Automation.

- *Source-driven development*, provides a greater agility to test out features with confidence (as all dependent packages are included (stable), and you can repeatedly prove the complete application life-cycle of your components).

#### 8.3.1 Expected Success Factors / KPIs

- Quicker delivery
- Less merge-conflicts
- Increased Business delivery
- Lower maintenance cost
- Reduction of footprint

## 9 CREATING AN UNLOCKED PACKAGE (DEV-HUB + SCRATCH ORGS)

---

1. Ensure the DevHub org is connected. The command below will list DevHub and Scratch Orgs.

```
sfdx force:org:list
```

- a. If DevHub is not connected use following command to login. Salesforce Login page will open prompting user to login. The user must login to an Org where Dev Hub is enabled.

```
sfdx force:auth:web:login -d -a DevHub
```

2. Go to the DX project directory and open the *sfdx-project.json* file. The bare minimum should look as follows:

```
{  "packageDirectories": [    {      "path": "force-app",      "default": true    }  ],  "namespace": "",  "sfdcLoginUrl": "https://login.salesforce.com",  "sourceApiVersion": "48.0"}
```

- a. If converting metadata to a DX project then first create a new DX project. Then go to that directory and run the convert command to convert the metadata to a DX project.

```
sfdx force:project:create --projectname <> sfdx force:mdapi:convert  
--rootdir <metadata_dir>
```

3. Create an Unlocked Package as follows:

```
sfdx force:package:create --name dreamhouse --description "My Package" --packagetype Unlocked --path  
force-app --nonamespace  
--targetdevhubusername DevHub
```

- a. --name is the package name. This name is an alias you can use when running subsequent packaging commands.
  - b. --path is the directory that contains the contents of the package.
  - c. --packagetype indicates which kind of package you're creating, in this case, unlocked.
4. The *sfdx-project.json* file should be updated as follows with version details and a package alias

```
{  "packageDirectories": [
    {
      "path": "force-app",
      "default": true,
      "package": "dreamhouse",
      "versionName": "ver 0.1",
      "versionNumber": "0.1.0.NEXT"
    }
  ],
  "namespace": "",
  "sfdcLoginUrl": "https://login.salesforce.com",
  "sourceApiVersion": "48.0",
  "packageAliases": {  "dreamhouse": "0Hxxx"
  }
}
```

5. When ready to release a package, a snapshot of it should be created called a package version. Prior to creating a version the *sfdx-project.json* file shown above can be updated with relevant version details.

6. The following command creates a package version:

```
sfdx force:package:version:create -p dreamhouse -d force-app -k test1234 --wait 10 -v DevHub
```

- a. -p is the package alias that maps to the package ID.
  - b. -d is the directory that contains the contents of the package.
  - c. -k is the installation key that protects your package from being installed by unauthorized individuals.
7. The package version creation process can take a few minutes. Once completed the output should look as follows:

```
Successfully created the package version [08cxxx]. Subscriber Package Version Id: 04txxx. Package
Installation URL:
https://login.salesforce.com/packaging/installPackage.apexp?p0=04txxx As an alternative, you can
use the "sfdx force:package:install" command
```

8. The package aliases section should have a new entry:

```
"packageAliases": {
  "dreamhouse": "0Hxxx",
  "dreamhouse@1.0.0-1": "04txxx"
}
```

9. The output of the package version creation command displays an installation URL. This URL can be used to download and install the package in a Salesforce org.
10. Packages have a *beta status* when they are initially created. Beta packages can't be installed in a production org. The following command can be used to *promote* a package to a production ready version:

```
sfdx force:package:version:promote -p dreamhouse@1.0.0-1 -v DevHub
```

11. When a production ready version is released that URL can be published in a central location used to download reusable components across the enterprise.

## 9.1 GENERAL COMMENTS ON UNLOCKED PACKAGES<sup>4</sup>

- Great for supporting parallel development tracks with different release schedules
- Used for support of Separation Of Concerns (SOC) for multiple LOBs
- Can be used to organize code by frontend/backend developer skills
- Have a package for shared components (see [limitation](#)).

# 10 APEX CROSS CUTTING CONCERNS EXAMPLE

---

This section covers how Apex Cross Cutting Concerns (ACCC) used unlock packaging. ACCC is broken down into three separate components:

## 1. Core ACCC

- 1.1. **ACCC Domain/Core functionality related to cross-cutting-concerns** – Custom Metadata, Apex Interfaces, etc.
- 1.2. **ACCC Common Classes** – Specific functionality support cross-cutting concerns (exception handling, logging, etc.)

## 2. Core PE/CDC

- 2.1. **PE/CDC Domain/Core functionality related to Platform Events and CDC** – Custom Metadata, Apex Interfaces, etc.
- 2.2. **PE/CDC Common Classes** – Specific functionality support platform events and CDC

## 3. Core Trigger Handling (TH)

- 3.1. **TH Domain/Core functionality related to Platform Events and CDC** – Custom Metadata, Apex Interfaces, etc. Due to time constraints, I added specific functionality support trigger handling. Note, at some point I will split this component to follow the above packages.

This allows a company (or individual) to **ONLY** load what they need. For example, **Core ACCC**, includes the base functionality. It does not include logging via Platform Events as this is specific to **Core PE/CDC**.

## 10.1 CORE ACCC PACKAGES

Core ACCC has two packages (***Accc\_Domain*** and ***Accc\_Common***)

---

<sup>4</sup> *Unlocked Packages(internal Salesforce)* – David Hogg

```

"packageDirectories": [
  {
    "path": "accc_domain",
    "default": false,
    "package": "Accc_Domain",
    "versionName": "ACCC Domain - Spring '20",
    "versionNumber": "0.5.0.NEXT",
    "definitionFile": "config/project-scratch-def.json"
  },
  {
    "path": "accc_common",
    "default": false,
    "package": "Accc_Common",
    "versionName": "ACCC Common - Spring '20",
    "versionNumber": "1.2.0.NEXT",
    "definitionFile": "config/project-scratch-def.json",
    "dependencies": [
      {
        "package": "Accc_Domain",
        "versionNumber": "0.5.0.LATEST"
      }
    ]
  }
],

```

## 10.2 CORE PE/CDC PACKAGES

Core PE/CDC has two packages (*Accc\_PE\_Domain* and *Accc\_PE*) with dependencies on Core ACCC components.

```

"packageDirectories": [
  :
  {
    "path": "accc_pe_domain",
    "default": false,
    "package": "Accc_PE_Domain",
    "versionName": "ACCC PE Domain - Spring '20",
    "versionNumber": "0.5.0.NEXT",
    "definitionFile": "config/project-scratch-def.json"
  },
  {
    "path": "accc_pe",
    "package": "Accc_PE",
    "versionName": "ACCC Platform Events - Spring '20",
    "versionNumber": "1.0.1.NEXT",
    "default": false,
    "definitionFile": "config/project-scratch-def.json",
    "dependencies": [
      {
        "package": "Accc_Domain",
        "versionNumber": "0.5.0.LATEST"
      },
      {
        "package": "Accc_Common",
        "versionNumber": "1.2.0.LATEST"
      },
      {
        "package": "Accc_PE_Domain",
        "versionNumber": "0.5.0.LATEST"
      }
    ]
  }
],

```

## 10.3 CORE ACCC TH (TRIGGER HANDLING)

**Core ACCC TH** has two packages (**Accc\_TH**) with dependencies on **Core ACCC** and **Core PE/CDC** components. Note, due to time constraints I did not split this package as done with the previous components.

```
"packageDirectories": [
  :
  {
    "path": "accc_th",
    "default": true,
    "package": "Accc_TH",
    "versionName": "ACCC Trigger Handler - Spring '20",
    "versionNumber": "0.6.0.NEXT",
    "definitionFile": "config/project-scratch-def.json",
    "dependencies": [
      {
        "package": "Accc_Domain",
        "versionNumber": "0.5.0.LATEST"
      },
      {
        "package": "Accc_Common",
        "versionNumber": "1.2.0.LATEST"
      },
      {
        "package": "Accc_PE_Domain",
        "versionNumber": "0.5.0.LATEST"
      },
      {
        "package": "Accc_PE",
        "versionNumber": "1.0.0.LATEST"
      }
    ]
  }
]
```

## 11 SUMMARY

---

Carving up an Org's Happy Soup *evolves* over time. The process will **NOT** be perfect the first time. If one makes a package too big initially, teams may collide (merge-conflicts) and/or smaller team to maintain. In addition, as mentioned, not all components can be broken into packages; however, as Salesforce DX evolves, the opportunity to refactor will become possible.

Finally, over time patterns will reveal themselves and provide an opportunity to create a common package shared by other packages. For example, large Orgs have similar SOQL access (Data Layer) to SObjects (Account, Contact, Leads, etc.) which can be factor into a common **Selector** Package.



## 12 REFERENCES

---

- *Trailhead introducing DX and Unlocked Packages* -
  - [Build Apps Together with Package Development](#)
  - [Package Development Model](#)
  - [App Development with Salesforce DX](#)
  - [CI Using Salesforce DX](#)
- *Dreamforce Recording* - [How Packaging Transformed AppDev at CarMax](#)
- *Dreamforce Recording* - [Advanced Techniques To Adopt Salesforce DX Unlocked Packages](#)
- *Working with Modular Development and Unlocked Packages* -- [Article](#)
- *Unlocked Packages [slicing] (internal Salesforce)* – Abhishek Sinha
- *Unlocked Packages(internal Salesforce)* – David Hogg
- *Pattern for Creating Reusable Components* – Rasika Wickramanayake

## 13 APPENDIX: NOTES

---

Below are notes to help avoid gotchas that may arise during use of scratch orgs, package creation and promotion, etc.

### 13.1.1 Check if Metadata is covered

When you start packaging (i.e. break-up a package into smaller chunks), you will need to ensure it is supported via the Metadata Dependency Coverage [here](#).

### 13.1.2 Using sfdx command force:source:{push|pull}

Both these commands *push* | *pull*, respectively, act on a Scratch Org and uses the *sfdx-project.json* to push/pull into a scratch Org. However, sometimes this pushes more than you want or need. You can either use *.forceignore* or create a temporary *sfdx-project.json* without the other packages but including the package dependencies. It is an inconvenience at this time.

### 13.1.3 Once your DevHub is setup , ensure DevHub on local machine is known

Scratch Orgs can only be created if there is a DevHub. You can ensure that the DevHub is properly registered by issuing this command line option:

```
sfdx force:auth:web:login -d -a <alias-name-for-dev-hub>
```

### 13.1.4 Scratch Orgs are temporary

Scratch Orgs represent a moment in time regarding your project. These Orgs should be used quickly ( 1-2 days) and deleted. This helps to ensure your project is CI-Ready and all tests pass (meeting code coverage). Also, helps eliminate the possibility that you run out of Active Scratch Orgs.

### 13.1.5 Using force:package:version:create

Package version creation takes place in a scratch org, whose configuration **must meet the minimum requirements for your package's metadata** in order for it to be deployed there. If it does not you can get an ERROR, such as:

```
ERROR running force:package:version:create: work: Total allocated capacity [3] exceeds the org limit [0]
```


Thus, specify the **-f** option for your Scratch Org (assuming you have specific features). For example, something like this:


```
sfdx force:package:version:create -p MyPackage -d force-app -k tst12 --wait 10 -v DevHub -f config/project-scratch-def.json
```

Then, the version creation org will have the same configuration as your project's regular scratch orgs, allowing your source to be deployed there. However, you first need to ensure the metadata components are packageable (see [Metadata Coverage Report](#)).

### 13.1.6 Unlocked Packages

Installed Unlocked Packages allow you to have a namespace as well as provide a warning about change.

 This Apex Class is part of a package. You'll lose any edits you make directly in the org if you reinstall or upgrade the package. Immediately inform your development team of any changes you make.

Note, this symbol, , which denotes an installed Unlocked Package. While a System Administrator can make changes to an Unlocked Package, there needs to be **Governance** to ensure those changes are communicated back to the Development Team. Otherwise, subsequent updates overwrite previous changes.

#### 13.1.7 Clean-up .SFDX directory within project

*Make sure you know how to find **Org** configurations for your project. They're listed in the '.sfdx' folder (this may be hidden on a Mac) contained within your main project directory. In that .sfdx folder, you'll see another folder called '**orgs**'. Here, you'll see every scratch org you've ever created and associated with your project listed. The files are labeled according to the system-generated scratch org username (like test-xmqpv1tmshae@example.com). You'll need to delete org configurations from time to time, in order to correct problems with `force:source:push` and `force:source:pull` (More on that below.) Be careful about modifying anything else in the .sfdx folder. Those files are intrinsic to the behavior of the CLI on your machine.*<sup>5</sup>

---

<sup>5</sup> From 'Working with Modular Development and Unlocked Packages' -- [article](#)

## 14 APPENDIX: LIMITATIONS FOR SALESFORCE DX

---

This [link](#) provides current limits for Salesforce DX. For example, the followings are some known limits. The link above provides the complete list.

### 14.1 SFDX PUSH FAILS WHEN MULTIPLE PACKAGES REFER TO THE SAME CUSTOM OBJECT

**Description:** When two or more package directories in the same 'SalesforceDX' project contain references to the same custom object, a source push will fail to push the changes from the second package

**Workaround:** This is currently a limitation of the packaging framework and the metadata API. The same object cannot be referenced in multiple *packageDirectories*. If multiple *SalesforceDX* packages are required to manipulate the same Object, then it is necessary to break those packages into separate projects.

### 14.2 CAN'T IMPORT RECORD TYPES USING THE SALESFORCE CLI

**Description:** We don't support RecordType when running the `data:tree:import` command.

**Workaround:** None.

### 14.3 LIMITED SUPPORT FOR SHELL ENVIRONMENTS ON WINDOWS

**Description:** Salesforce CLI is tested on the Command Prompt (`cmd.exe`) and Powershell. There are known issues in the Cygwin and Min-GW environments, and with Windows Subsystem for Linux (WSL). These environments might be tested and supported in a future release. For now, use a supported shell instead.

**Workaround:** None.

### 14.4 THE `FORCE:APEX:TEST:RUN` COMMAND DOESN'T FINISH EXECUTING

**Description:** In certain situations, the `force:apex:test:run` command doesn't finish executing. Examples of these situations include a compile error in the Apex test or an Apex test triggering a pre-compile when another is in progress.

**Workaround:** Stop the command execution by typing control-C. If the command is part of a continuous integration (CI) job, try setting the environment variable `SFDX_PRECOMPILE_DISABLE=true`.

### 14.5 DEV HUB AND SCRATCH ORGS

#### 14.5.1 Salesforce CLI Sometimes Doesn't Recognize Scratch Orgs with Communities

**Description:** Sometimes, but not in all cases, the Salesforce CLI doesn't acknowledge the creation of scratch orgs with the Communities feature. You can't open the scratch org using the CLI, even though the scratch org is listed in Dev Hub.

**Workaround:** You can try this workaround, although it doesn't fix the issue in all cases. Delete the scratch org in Dev Hub, then create a new scratch org using the CLI. Deleting and recreating scratch orgs counts against your daily scratch org limits.

#### 14.5.2 Error Occurs If You Pull a Community and Deploy It

**Description:** The error occurs because the scratch org doesn't have the required guest license.

**Workaround:** In your scratch org definition file, if you specify the Communities feature, also specify the Sites feature.