DX Packaging

Putting it all together with Unlock Packages

Bill Anderson

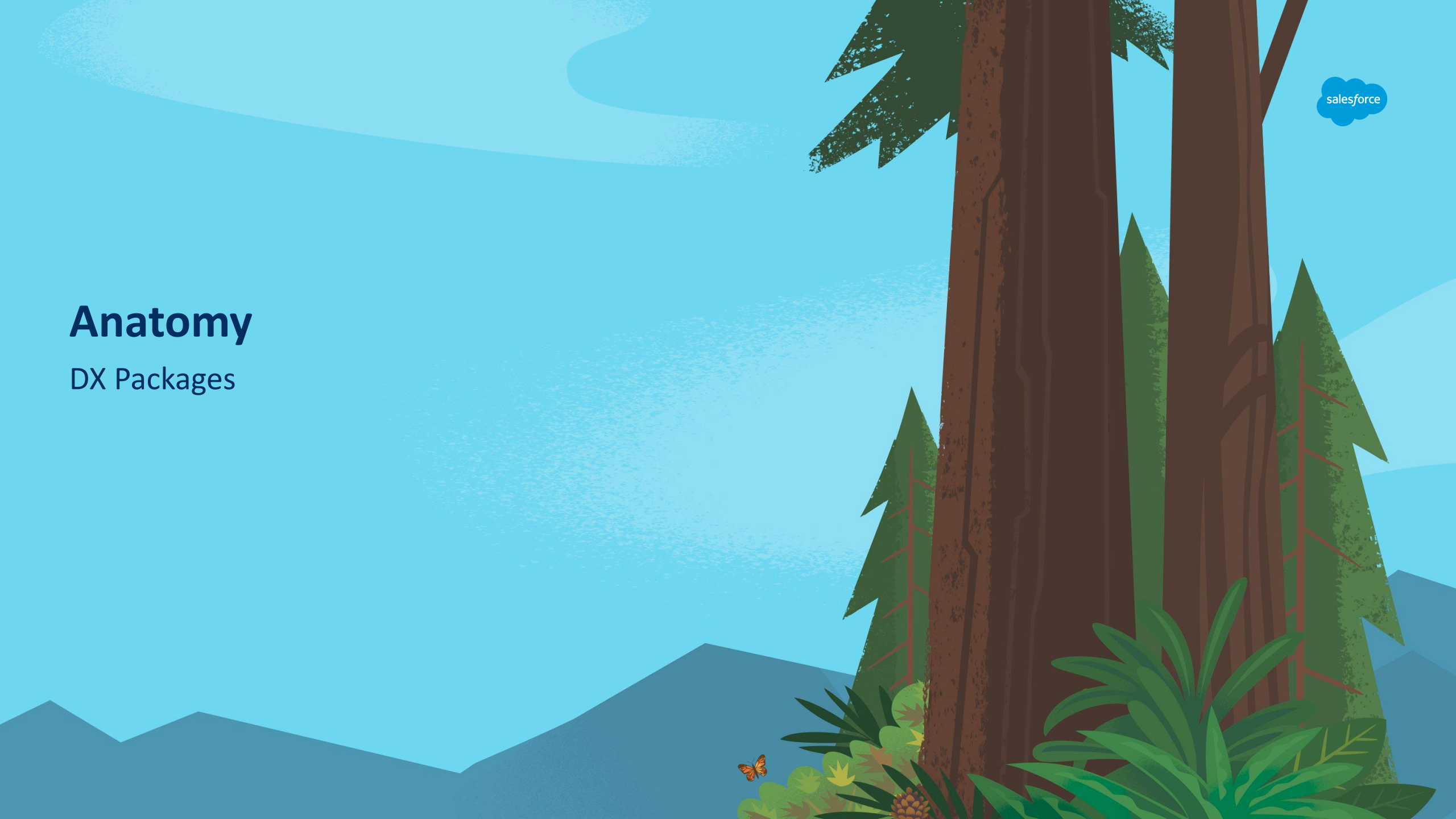
March 11, 2021

Topics

1. Anatomy
2. Benefits
3. Guidelines

Anatomy

DX Packages



DX Packaging

Why?

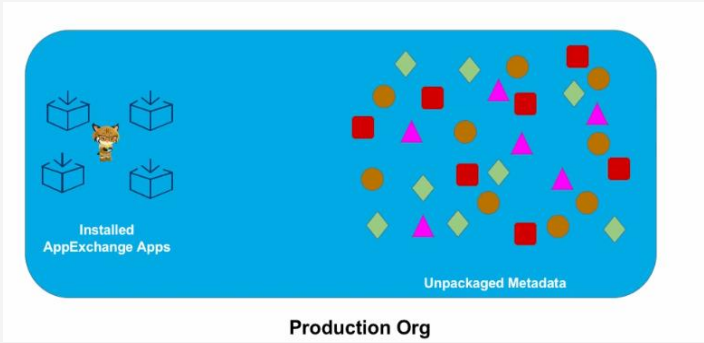
Previously, Salesforce Orgs contain various components (lwc, aura, objects, list views, profiles, PermissionSets, etc.). Often these components existed in singularity or earlier packaging mechanisms.

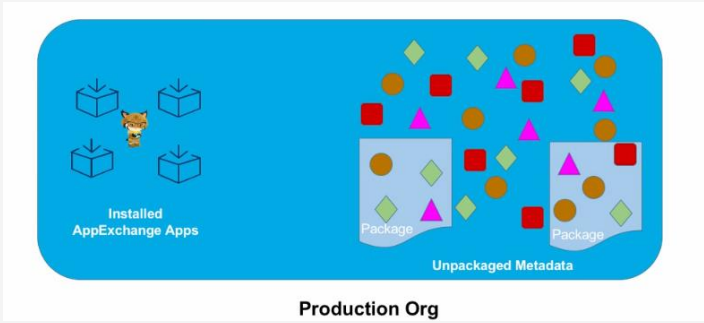
Transitioning these singular components from Sandbox to Sandbox was prone to errors as each Salesforce Org was its own source of truth.

Managing these singular components via Unmanaged Packages or 1st Generation provided a better handle yet was not ideal (still having Salesforce as the source of truth)

Fast forward to today, we have 2nd Generation packaging to support a better way to control, share, deploy and reuse functionality via a single source of truth (i.e., GIT Repository).

With DX Packaging, we have an Anatomy that allows us to narrow the scope of components into unique durable components from a single source of truth.

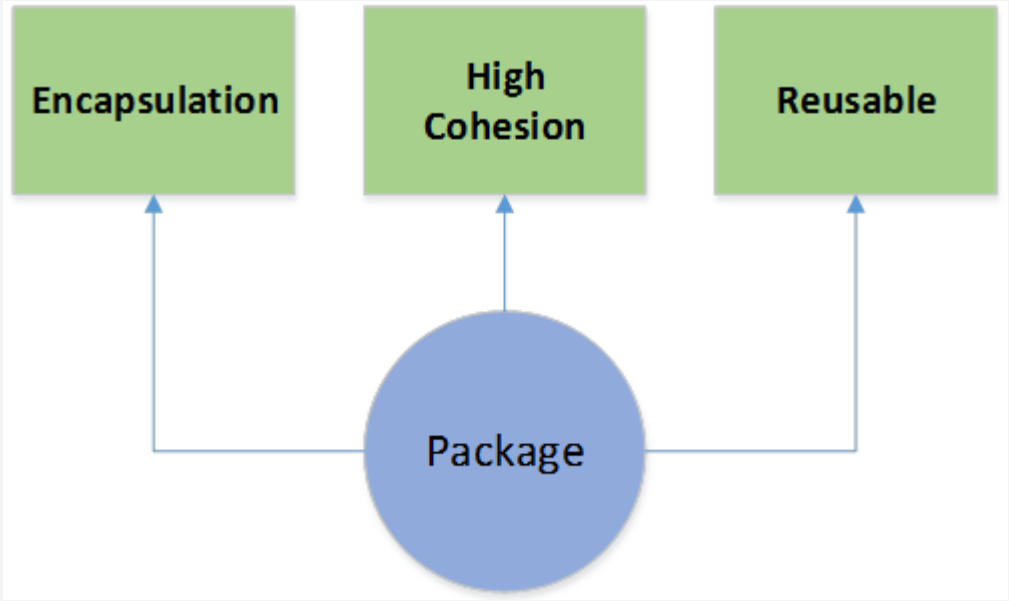




DX Packaging – What is it?

Anatomy

DX Packaging, much like libraries found in other languages (i.e., C#, Java, C++, etc.), maintained in a Git Repository can be encapsulated, deployed, shared and reused. With DX Package, there exist Three Pillars, *Encapsulation*, *High Cohesion* and *Reusability:*

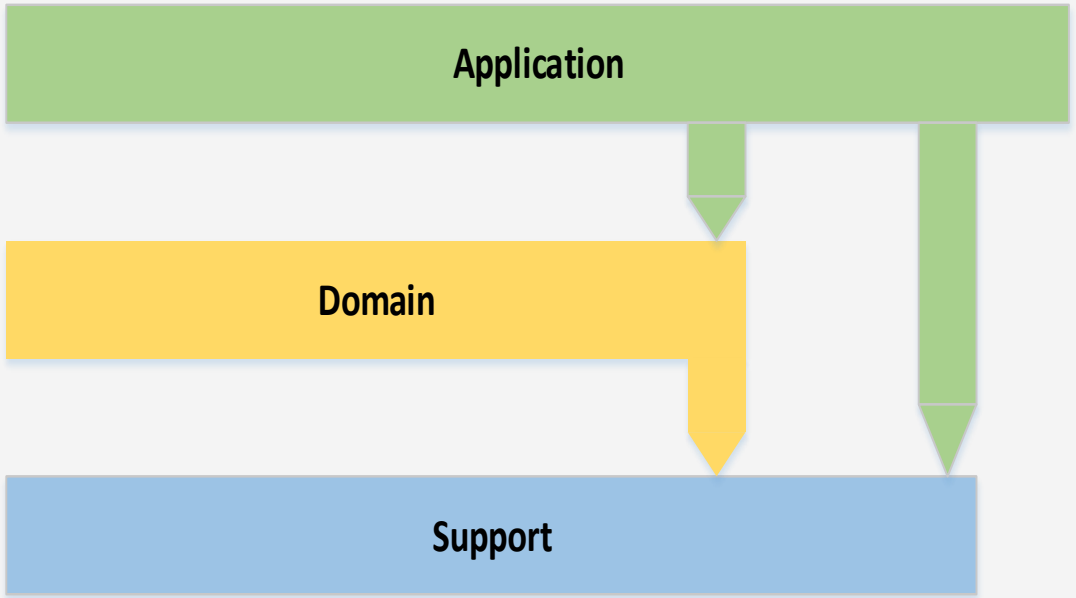


Pillars of a Package	
Encapsulation	Defines the bundling of data with the methods that operate on said data. Encapsulation is used to hide the values or state of a structured data object inside a package
High Cohesion	Defines how closely all the components in a package support a common purpose.
Reusable	The package can be reused without consequences of the environment; morphing to each environment.

DX Packaging – Categorizations

Managing the Packages

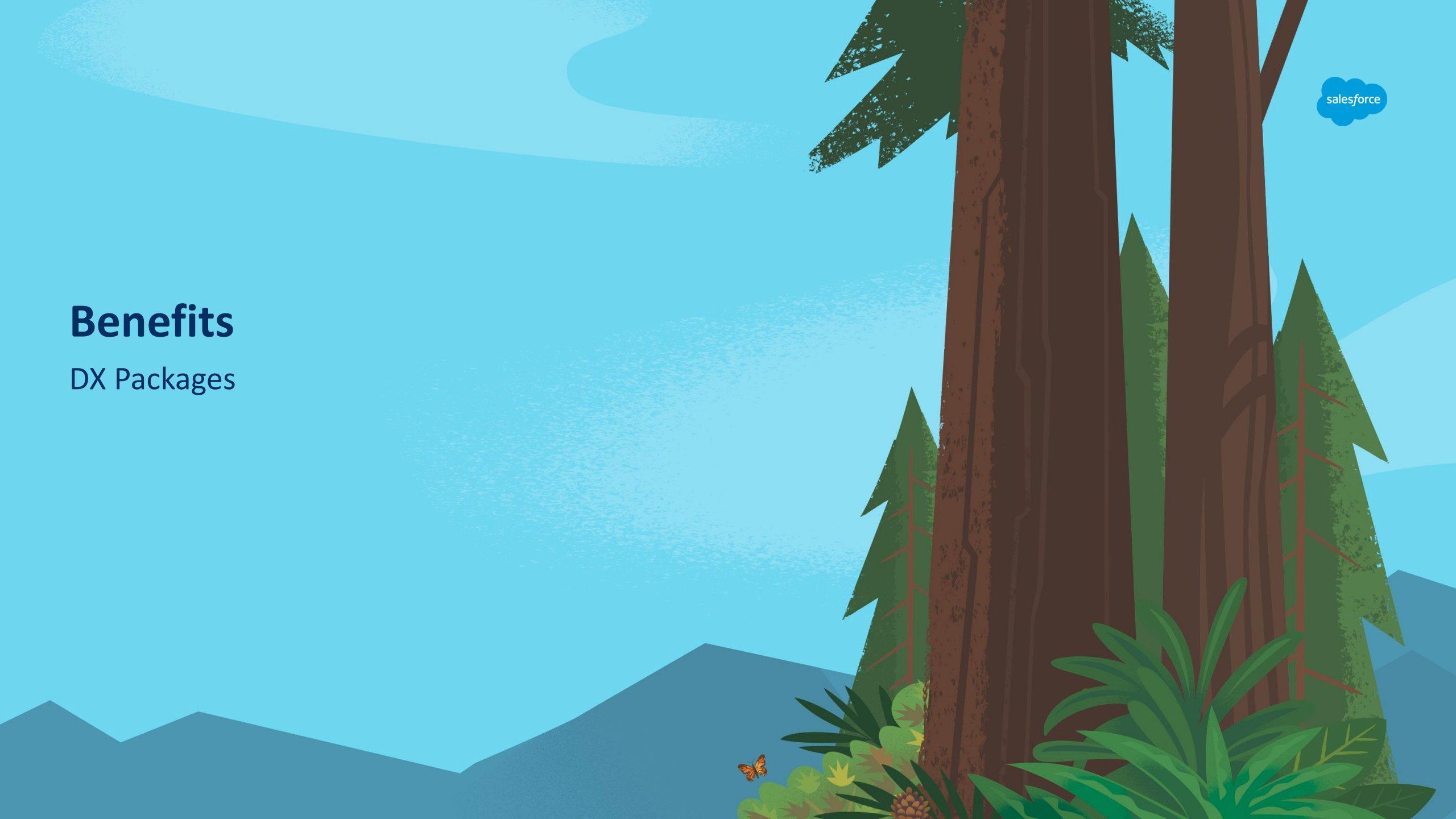
Understanding the core pillars of a DX package, we can now begin to classify. Packages fall into three categories, *Application*, *Domain* and *Support*. These base concepts follow the same paradigm found in other Frameworks:



Package Categories	
Application Package	Encapsulate expertise in a program such as Leads and Referrals, Customer Management, etc. These high-level packages may utilize both Domain and Support packages
Domain Package	Encapsulate expertise to a domain. These packages can provide a horizontal slice, such as, trigger, data management, etc.
Support Package	Encapsulate expertise to specific functionality, such as communication, data manipulation, data access, etc. These support packages help support **Domain** and **Application** packages.

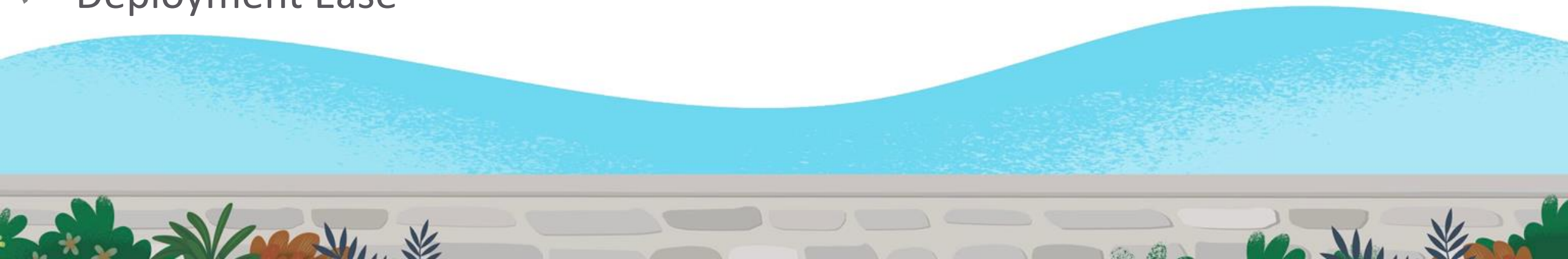
Benefits

DX Packages



Packaging Benefits

- ✓ Bootstrap's development or application process
- ✓ Eases Debugging and Application Maintenance
- ✓ Reduces Customization Length
- ✓ Deployment Ease
- ✓ Maintained and Controlled in Single Source Of Truth
- ✓ Versioned
- ✓ Single Responsibility
- ✓ Focused Testing



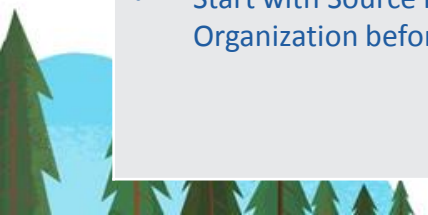
Guidelines

DX Packages

Packaging Guidelines

Guiding Principles

Guidelines	Benefit	Comments
• Keep Packages Highly Cohesive (Single Responsibility)	• Easier to Test, Maintain and Support • Reduced Bulk/Footprint • Deployment and Upgrade Ease	Packages should support a single concept. Cross-Cutting Concerns, Separation of Concerns, Dependency Injections, Leads and Referrals, etc. Thus, frameworks like FFLIB, ACCC, Force-DI, etc. represent **Support** Packages which **exist in their own DX Package [Single Responsibility]**
• Dependency Inversion	• Supports change • Allows implementation to change without change to behavior. • Less Brittle	If a package needs the ability to morph to its environment, provide that either through abstraction and/or dependency injection. Business Processes change; thus, prefer to *change without disruption*.
• Use nomenclature that supports your business model	• Identity • Categorization • Management	Define a nomenclature which adequately defines their Category (i.e., Application [*app*], Domain [*dom*], Support [*sup*]), Ownership and use). Prefixes are at most 15 characters; for example, *sup_wim_lead, dom_crv_lead, app_wf_lead.* Please note, namespaces are a better approach to avoid collision and improve organization.
• Versioning (with SOT)	• Identity and Maintenance	Use the Repository to tag your release. Allows correlation back to your specific package. Supports traceability and maintenance
• Depend on Knowns or Mocks	• Environment Agnostic • Loose Coupling	Do not depend upon items in which you have no control. For example, a unit test should not depend upon fields in an object which may or may not exist. Use MOCKs to validate. Explicit Dependencies should be internal; implicit dependencies should be configurable. If there must be knowns, consider splitting into a separate Support Package.
• Start with Source Format Organization before packaging	• Source format organization allows you to understand the relationships and dependencies visually w/o Packaging • Separates the layers/concerns into a logical dependency	Break up the source into logical order and dependency (*sfdx-project.json*). If you take a common simple SOC (i.e., Common, Business/Service, Data) approach in you source format you can first organize your packages accordingly. For example, cross-cutting concerns (Common) address utilizes (String, Exception Handling, Security, Logging, etc.); of which, may result into other packages.




Thank
You