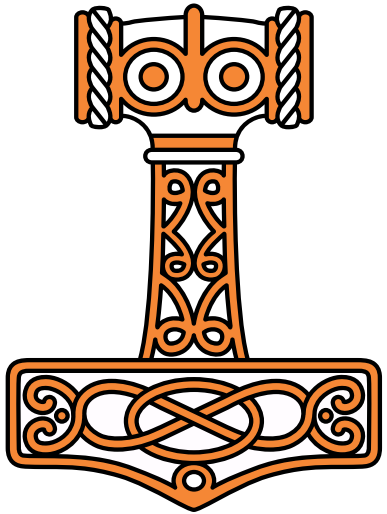
DYALOG

Olhão 2022

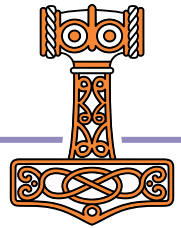
Creating, Maintaining and Publishing APL Packages



Josh David
Brian Becker

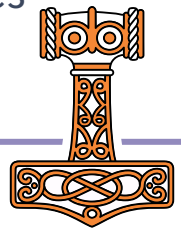
Agenda

- Session 1
  - Why and What
  - APL Packages
  - Exercise 1 – Introduction to Tatin
- Session 2
  - More Tatin
  - Exercise 2 – create a package with dependencies
- Session 3
  - Recommended Practices
  - Workflow



Introductions and Goals

- Introductions
- Goals for this Workshop
  - Understand what a package is in general and in an APL context
  - Learn a bit about Tatin – an APL package manager
  - Build and publish a package with dependencies
  - Understand recommended practices
  - Learn a bit about workflow and Dado
  - Leave energized and invigorated to create and publish your own packages



Building a Treefort

- Buy a hammer, or better yet, borrow one
- Buy some standard dimensional lumber
- Buy some standard fasteners (nails, etc)
- Buy or borrow a ladder
- Build the treefort

Building a Treefort the APL way

- Dig in the ground for iron ore
- Smelt the ore to fashion your own hammer, nails and saw
- Use the saw to cut down a tree and fashion your own lumber
- Build your own ladder
- Build the treefort

Building a Treefort the APL way

How many of us have written (at least) one set of utilities to

- manipulate character data (delete_blanks, center, etc)?
- work with dates (daysdiff, day_of_week, etc)?
- manage data on file (openfile, exists, etc)?

Building a Treefort the APL way

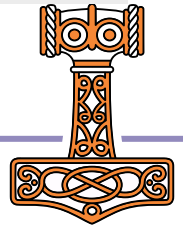
Packages on PyPI (2021)

337,215 projects 2,993,887 releases 5,100,558 files 548,125 users

The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.

Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.



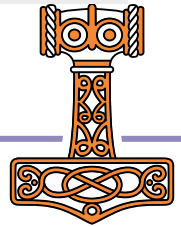
Packages on PyPI (2022)

406,065 projects 3,844,709 releases 6,852,067 files 628,580 users

The Python Package Index (PyPI) is a repository of software for the Python programming language.

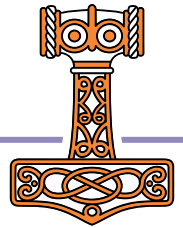
PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.

Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.



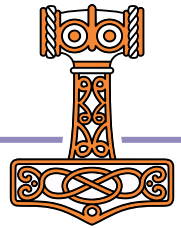
What is a Package?

- A software component that functions as a single entity to accomplish a task or a group of related tasks.
- Generally used as a unit within a larger application context.
- In APL, it could be a namespace, class, function, or a collection of them, plus non-APL assets (HTML files, shared libraries, etc)
  - Candidates include Jarvis, HttpCommand, subsets of dfns workspace, Dyalog Cryptographic Library, and others in the pipeline
- A package may have dependencies on other packages.



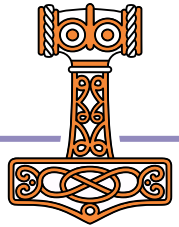
Why are packages important?

- Reduced duplication of effort.
- Leverage the work of others
- Expectation from APL newcomers who have experience in other environments.



Package Dependencies

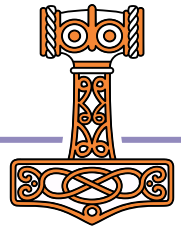
- A package may depend on zero or more other packages which may in turn depend on zero or more packages (which may in turn depend...)
- For instance
  - The package FilesAndDirs depends on the package
    - APLTreeUtils which depends on the package
      - OS which has no dependencies
- What should be done if a package is a dependency of more than one other package?
  - For instance
    - Many of the APLTree packages have a dependency on APLTreeUtils



Semantic Versioning

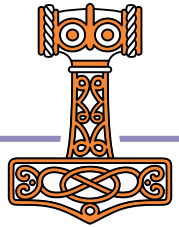
- In short semantic versioning is a means to consistently reflect the type of change to a package from one version to another.
- At a minimum semantic versioning consists of
  - A major version number, a minor version number, a build (or patch) number

2.11.23 – 2 is the major version, 11 is the minor version, and 23 is the patch number



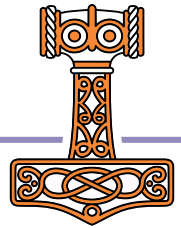
Semantic Versioning

- The build number is increased whenever a change that does not add feature or change existing behavior is made.
  - Bug fixes, Code refactoring, fix a typo in a comment
  - Users can safely upgrade to a newer build number
- The minor version number is increased whenever there is feature added, but no previous behavior is changed.
  - Users can safely upgrade to a newer minor version number
- The major version is increased whenever there is a change in existing behavior.
  - Different results for the same arguments
  - Users should review the changes before upgrading to a newer major version number.



Semantic Versioning

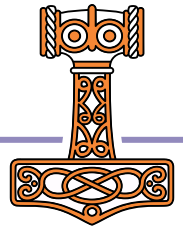
- Consider:
  - You change a function in your package to accept an additional optional element in the right argument.
  - Is this a major or a minor version number change?
    - Minor – because existing applications will continue to function the same way as before.



Assumptions

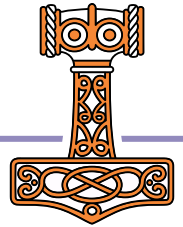
For creating packages, we'll make the following assumptions:

- Code in files
  Your APL source code is stored in text files rather than in a workspace
- Optional but recommended: Source Code Management
  Using a source code management system like git makes it easier to manage code revisions and history.



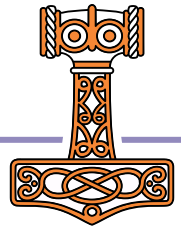
Tools to Develop/Manage APL Packages

- Tatin – https://github.com/aplteam/Tatin
  - The Tatin client allows you to incorporate APL packages and any dependencies they may have into your APL application.
  - Additionally, the client enables you to maintain and publish packages
  - A Tatin server exposes and hosts a registry where packages may be imported from or published to
  - You may run your own Tatin server, which exposes a or use the publicly available ones:
    - Production server: https://tatin.dev
    - Test server: https://test.tatin.dev
- Dado - https://github.com/the-carlisle-group/Dado/
  - A framework for helping manage and deploy APL projects
  - Centered around a GitHub-based workflow



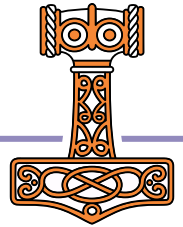
Exercise 1

- test.tatin.dev
- Install Tatin Client
- Tatin Tour
-]loadpackages HttpCommand
-]loadpackages FilesAndDirs



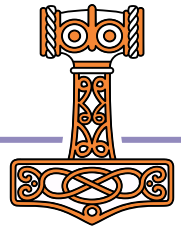
Tatin Servers/Registries

- The "official" Tatin registries may be found at:
  - Production: https://tatin.dev for publishing "production-level" packages
  - Test: https://test.tatin.dev – a sandbox to help you experiment with tatin and the creation/maintenance of packages
- The registries are essentially identical in functionality with following exceptions:
  - You must enroll with and receive an API key to use the production registry
  - Deletion policy:
    - Packages published to the production registry are never deleted.
    - Packages published to the test registry may be deleted
  - The test registry may occasionally be "reset"



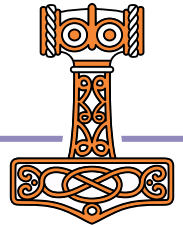
Install the Tatin Client

- Download the latest release from https://github.com/aplteam/Tatin/releases
- Unzip the contents into your MyUCMDs folder
- Once installed, the Tatin client is exposed through a number of user commands in the Tatin group.



Tatin User Commands

- Like all user commands, the Tatin user commands begin with a right bracket `]`
- They are found in the Tatin user command group
  ```
  ]tatin -?
  ```
- Unambiguously named commands do not require the Tatin group name
  - `]createpackage` is unambiguous and does not need the group name
  - `]version` could be either `]tatin.version` or `]tools.version` so you must include the group name.
- In addition to the documentation found on the Tatin registry sites, all Tatin user commands have standard user command help information. For instance:
  - `]createpackage -?`

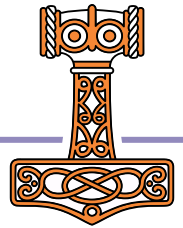


Querying Tatin Registries

- `]ListRegistries` will list all the defined Tatin registries

```
      ]ListRegistries
URI                      Alias       Port  Priority
-----------------------  ----------  ----  --------
https://tatin.dev/       tatin          0       100
https://test.tatin.dev/  tatin-test     0         0
```

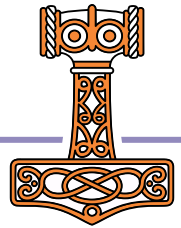
- A registry may be identified by its URI or an "alias"



Querying Tatin Packages

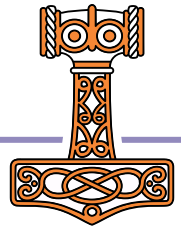
- `]ListPackages` will list all the defined packages for a Tatin registry

```
      ]ListPackages [tatin]
Registry: https://tatin.dev
Group & Name                  ∑ major versions
------------                  ----------------
aplteam-APLGit                               1
aplteam-APLProcess                           1
.
.
.
davin-DateTime                               1
davin-FilePlus                               1
.
.
.
dyalog-HttpCommand                           1
dyalog-Jarvis                                1
```



Creating and Publishing a Package

- First, write the code for the components of your package. ☺
  This could be as simple as a single function or it might consist of several namespaces, classes, external resources, etc.
- Organize your components into a folder structure or .zip file
- Specify the package configuration
- Specify any dependencies on other packages
- Publish your package

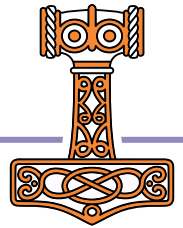


Specify the package configuration

- Define the package configuration file (apl-package.json) in the folder for the package:

```
]PackageConfig foldername -edit
```

  - This file may also be created/edited using the editor of your choosing

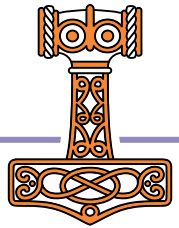


Specify any dependencies on other packages

- Define the dependencies file (apl-dependencies.txt) in the folder for the package:

```
]PackageDependencies foldername -edit
```

  - This file may also be created/edited using the editor of your choosing

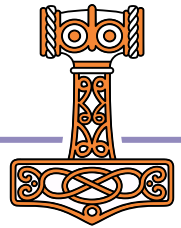


Publish your package

- To publish your package on a Tatin registry use:

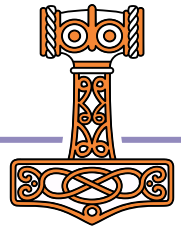
```
]PublishPackage foldername registry
```

  - registry is the URI or alias for the registry.



Package Maintenance

- Maintaining your package is simply a matter of publishing a new version
  - Be sure to increment your package version number, preferably using semantic versioning guidelines



About

```
]dado.about
```

```
  APL                        Dado
 \-----/       Dyalog APL Development Operations
  |   |                       ----
  |   |     https://github.com/the-carlisle-group/Dado
 /_____\                      ----
 |_____|       "Put your apps on a solid foundation"
```

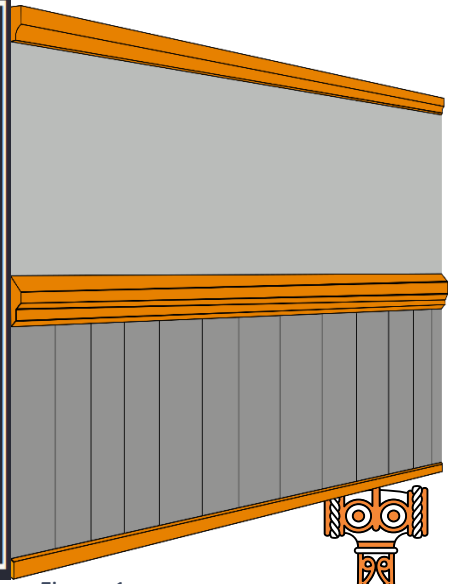
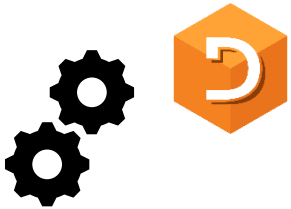


Figure 1

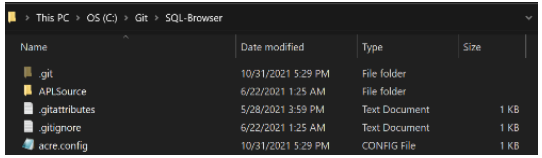
Definitions

Application



Package

Project



DadoFlow

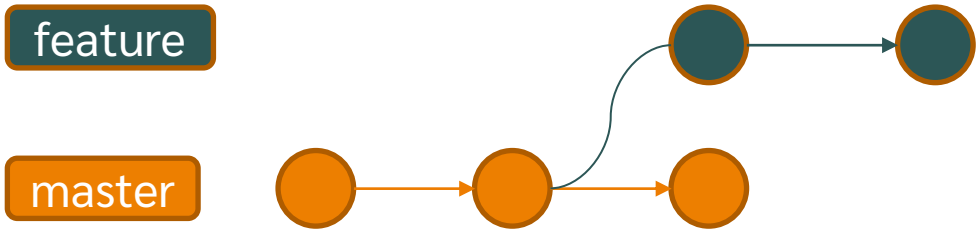
- What DadoFlow tries to avoid

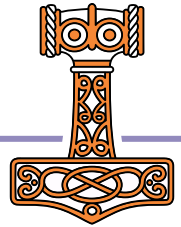
Figure 2

DadoFlow

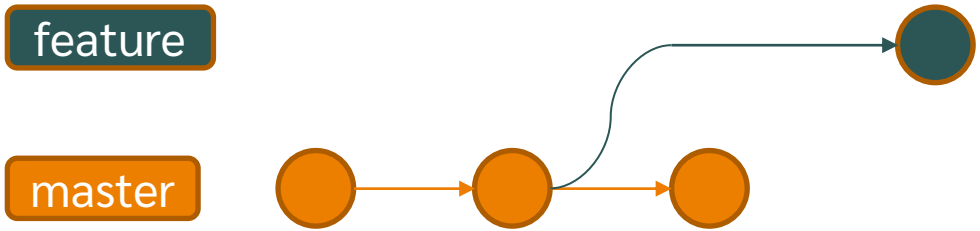
- What DadoFlow tries to avoid

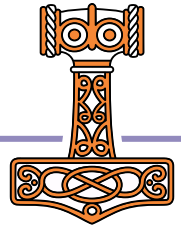
Linear History



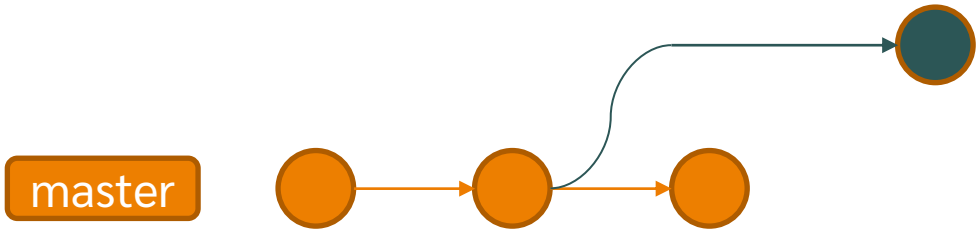


Linear History



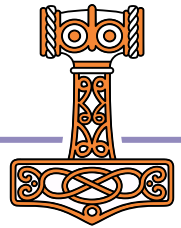


Linear History



Not a workflow for everyone

- Not the best choice if you:
  - Need to concurrently develop parallel branches (v2 and v3)
  - Are constantly requiring patches to old versions



Next Steps

- Better integration between Dado and Tatin
- Seed Tatin with more APL libraries