Developing Applications

Release 15.5 Service Level 45

Feb 02, 2023

Contents

CHAPTER 1

About component architecture

We came a long way: at the beginning of the 90's, CIM Database was a monolithic Unix workstation application, behaviour and business logic resided mostly in compiled code. But it already possessed limited capabilities for customizing. Those early beginnings have evolved into a fully featured development framework. Today, there is a "model-driven" design approach, the Python programming language, and possibilities to build web applications and services based on CIM Database. In fact, all of the business logic has been factored out from the application kernel and is developed using the means provided by the platform. On top of the technology platform, in-house application developers, customers, partners, consultants and 3rd-party service providers develop and customize applications:

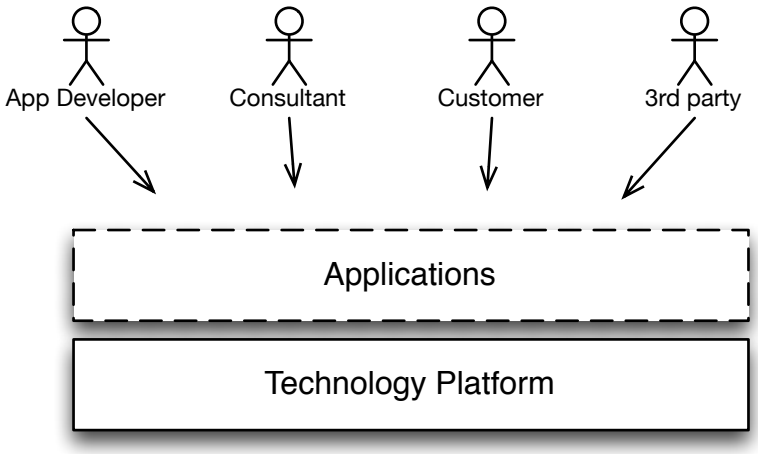

Fig. 1: Applications sit on top of the platform

- they design information models by describing entities, attributes and relationships in the data dictionary,
- they build user interfaces by describing dialogs, tables, menu items and others, by building web-based UI elements using standard web application development techniques like templating, CSS and Javascript,
- they tackle business logic, stand-alone tools, web-services and other programming tasks using Python and the extensive APIs of CIM DATABASE.

But something has been missing.

First, there has been no sufficient infrastructure to develop, test, distribute, install and upgrade *subsets* of the application level functionality of CIM Database. Applications either ship relatively infrequently with releases of

the technology platform, or by utilizing complex, scripted install procedures. Methods for developing, upgrading and testing applications vary wildly between teams and projects without converging into a set of easy to use best practices.

Second, customizing, the act of adapting the pre-packaged standard to customer needs, has always been partially destructive in nature. While we introduced means to separate customer-specific Python code from code we shipped, adapting that part of applications described in the data dictionary or the UI catalog, was destructive: changing a dialog or adding a field to a class meant to modify the original description of that item, loosing the original. We've taken explicit measures to leave those parts of applications alone during updates. This effectively lead to a situation where customers are stuck with their customized solution and have difficulties in reaping the benefits of new software releases.

The component architecture of CIM Database 10 solves both problems. We have introduced means to handle *all* components of an application module, i.e. the information model (schema), user interface, program code and assets like templates and icons, in an orderly manner throughout the module's lifecycle, and separate from other modules. We have build mechanism to aid application developers in collecting application content and serializing it to disk. We have adopted packaging and distribution best practices of the Python community, giving opportunity to deploy battle-hardened tools and the broad expertise of that community. And we have found ways to non-destructively separate customized application content from the original content shipped with a release that leaves existing applications and customizings intact and allows painless migration for pre-10 users.

CHAPTER 2

Overview

We use "application content" to term everything that makes up a CIM DATABASE application. Application content is handled by application developers during the lifecycle of an application, distributed to customers and deployed. Application content may be:

- data dictionary declarations, and thus, indirectly, database schema objects
- UI elements like menu items, dialogs, tables, messages, catalogs, color definitions, etc.
- application assets like images, icons, templates etc.
- definitions for reports and metrics, roles, groups and the access control system
- documentation
- Python code
- default settings

The new approach of CIM Database 10 is to organize *all* application content into a set of *modules*. Modules bundle the different kinds of application content that logically belong together. As an example, imagine a CIM Database class together with it's attributes, the dialogs used to create and edit objects of that class, icons, and the Python code containing the business logic for the class.

Modules have a unique name and provide a namespace for their content items.[1] All module names start with a "vendor prefix", followed by a dot and the module name. CONTACT's vendor prefix is `cs` (for "CONTACT Software"). Typical module names will be `cs.documents`, `cs.pcs.tasks` and so on. Customers and 3rd-party developers will each pick a unique vendor prefix for their own work, typically much longer than the two characters of `cs`. A customer named "Froobuzz" might choose `froobuzz` as their prefix.

The fact that module names look like Python identifiers is not by accident: the Python code belonging to a CIM Database module lives in exactly that namespace – and application content other than code is stored in the same filesystem location where the Python package resides. A significant element of the component architecture are tools to store database content into a module's filesystem structure during development, and restore it from there into an instance database for deployment. On disk, a module therefore looks like an ordinary Python package, except there are additional folders inside the package directory that store non-code application content, like database content, images, icons, documentation and so forth. This enables reasonable version control procedures and easy packaging and distribution of modules.

Modules are the atomic building blocks of CIM Database applications and as such normally are rather small – too small to handle each module's lifecycle, distribution and installation separately. We therefore introduced the

[1] This is not quite true yet: because for now we didn't break structures and conventions for identifying database items, most database content has kept it's primary key structure. Classes, for example, must still have globally unique names in this release.

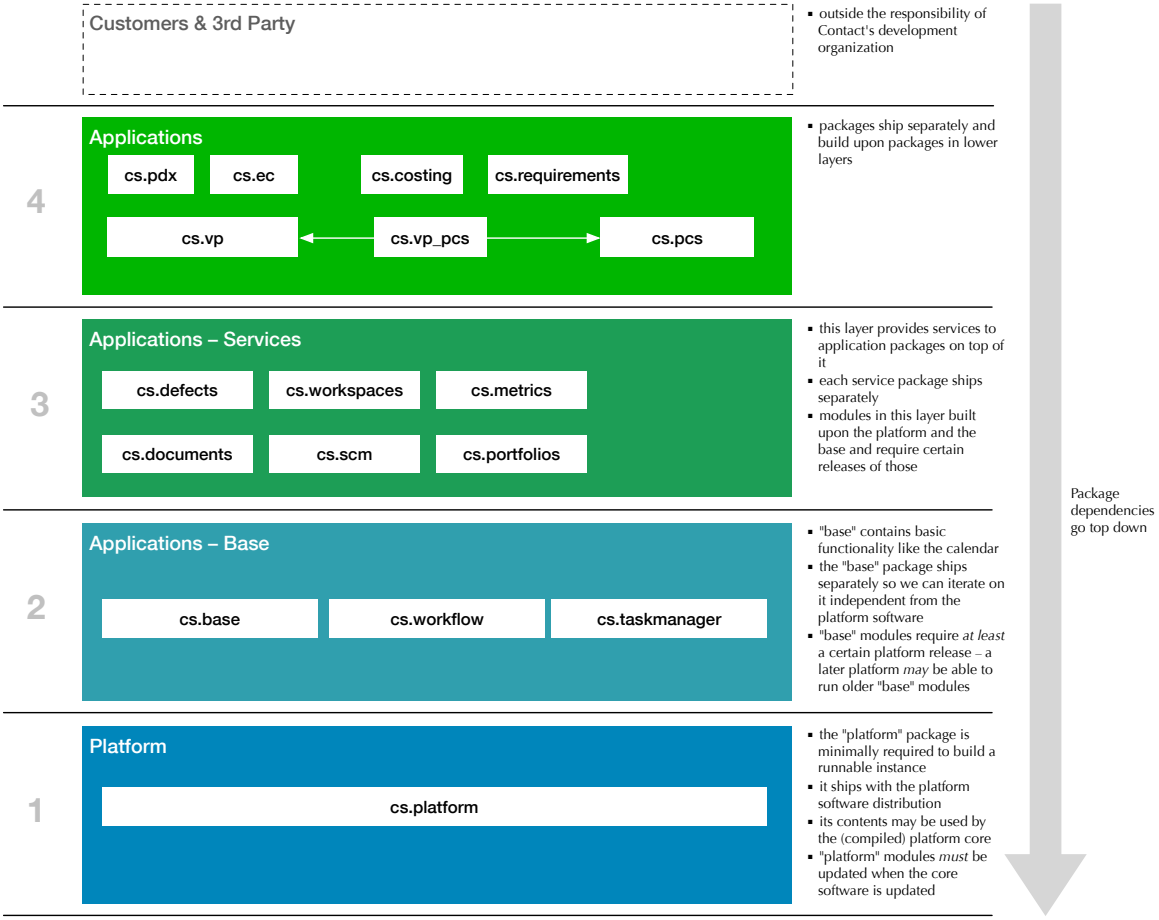

Fig. 1: Package organization into layers

additional concept of *packages*, that bundle a set of related modules together. CIM Database packages exactly resemble Python's installable packages or "distributions". Packages are what developers work on, what gets released and packaged into a distribution and is deployed into end-user systems. Naming conventions are very similar to modules, starting with a vendor prefix, followed by a dot and the package name. Most CONTACT Software packages are named after their "main" module, for example *cs.costing* (the package) consists of `cs.costing` (the module) and two other modules `cs.pcs.projects_costing` and `cs.costing.documents`.

Naturally, application content in modules and packages is not totally isolated: classes in one module are related to other classes, code imports functions from elsewhere, and modules re-use assets from other packages. To support the component architecture tools, modules have relationships like *uses*, *extends* etc. to each other. Those relations are maintained during development. Packages on the other hand have just one simple relation: packages may *require* the existence of other packages in the environment into which they are to be installed. Requirements are declared in the package meta-data and form a directed, acyclic dependency graph (whereas module relationships might be much more complex). Python tools for installation and deployment automation use those requirements to automatically choose and install consistent and runnable package configurations.

An overview of all the "*cs*" packages available with the initial release of CIM Database 10 is shown in figure *Package organization into layers* (page 4). In this illustration all "*cs*" packages are classified into layers that denote differing responsibilites and a hierarchy: packages in one layer may only require those in the same or lower layers.

2.1 Customizing

Customizing involves adapting standard applications to the needs of a specific customer. This has traditionally been done by adding Python code and other file-based assets to the "instance directory", and by destructively modifying database content, such as data dictionary and user interface declarations. Under the new approach of CIM DATABASE 10, customizing is handled the same way as application development: Python code, entity declaration, user interfaces etc. are bundled into custom modules and packaged in custom packages. This enables lifecycle management best practices for customizations and custom development.

Customizations are built the same way as before in CIM Database: custom code can connect to signals raised by the framework (preferred), or derive from application-level Python classes. Database content can be modified "in-place" to fit the requirements. The main difference is the fact, that the original data is still available in the on-disk copies of the application packages. This way, for example a dialog originating from an application can be modified as needed, but CIM Database will still be able to compute the difference between the modified dialog and the original. This difference is then stored as a *patch* inside the custom package and can be lifecycle-controlled, packaged and deployed into a target system.

2.2 Implications and advantages

The component architecture will have wide reaching consequences for the portfolio of solutions available for the CIM Database platform. The change is driven by enabling separate life-cycles for components and by enabling product variants through optional and alternative packages.

- We gain the ability to iterate independently and more quickly on single packages. This enables us to focus resources and increase development pace for areas demanding improvements.
- We enable product management to build separate product bundles out of existing components and customers to set up configurations tailored for their needs.
- We can build vertical industry specific solutions by providing alternative packages.
- Customers and 3rd-parties are enabled to develop packages on their own, thus possibly spawning an ecosystem of CIM Database "apps".
- We can start to provide alternative implementations of foundational packages to fix long-standing design faults that hinder innovative approaches, while keeping support for legacy implementations still in use by customers.

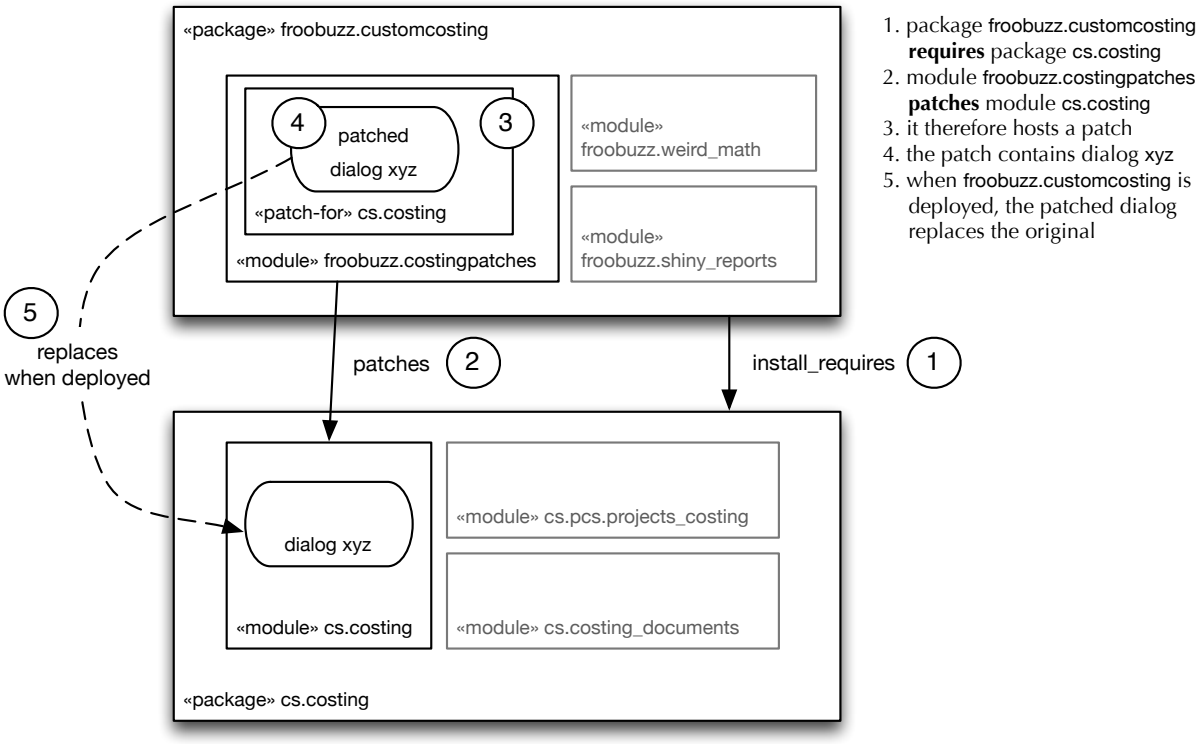

Fig. 2: How patching works

Component architecture also improves methodes and tools available, and suggests best practices: developers and deployers get a structured way to deploy packages, merge updates and transport changes.

CHAPTER 3

Inside modules and packages

3.1 Packages

Figure *On-disk layout of packages and modules* (page 8) shows a developer's view into the filesystem layout of a package. Each package has a top-level directory. Inside the package directory modules live in a namespace hierarchy. Module directories contain all module assets, specifically also database content, stored as `*.json` files inside the `configuration/` directory.

The top-level package directory also hosts package-wide files and directories for development automation etc., the most important being `setup.py` used by Python's "setuptools" for packaging and installation and `buildout.cfg` used by the build automation tool "zc.buildout". A simplified `setup.py` might look like[1]:

```
from cdb.comparch.pkgtools import setup

setup(
  name = "cs.costing",
  install_requires = ['cs.documents'],
  cdb_modules = [
    "cs.costing",
  ],
  cdb_services = [],
  docsets = [
    "doc/userman/de",
  ],
)
```

The other important package configuration file is `buildout.cfg` which is used by "zc.buildout" to automate the setup of development environments. A simple `buildout.cfg` looks like:

```
[buildout]
newest = false
develop=.
parts=inst
eggs = cs.costing
```

(continues on next page)

[1] Note the `setuptools` wrapper imported from `cdb.comparch.pkgtools`. This wrapper simplifies `setup.py` for application packages. The `setuptools` documentation is recommended reading for all developers. We use the `setuptools`-fork `distribute`. Documentation is at http://pythonhosted.org/distribute/.

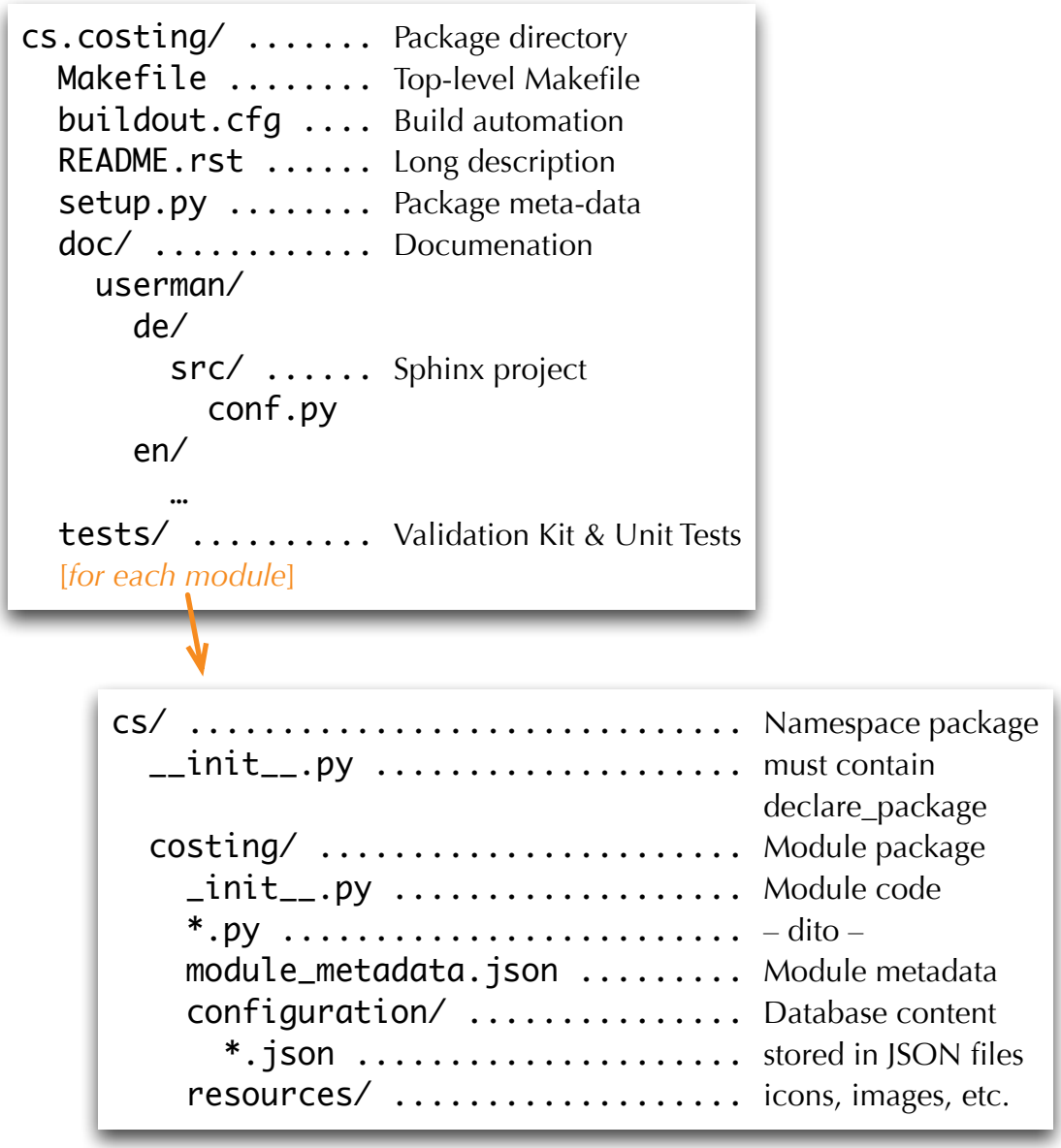

Fig. 1: On-disk layout of packages and modules

(continued from previous page)

```
[inst]
recipe=cs.recipe.instance
namespace=cs
eggs=${buildout:eggs}
```

`cs.recipe.instance` is a buildout recipe that creates a CIM Database instance. It is intended to support application development by adding repeatable and automatic instance creation to package development environments controlled by "buildout". It will do four things:

1. Create an instance directory named after the part name `dev_inst` inside your buildout.
2. Create scripts in `bin/` that wrap all CIM Database commands that require an instance to run (`powerscript`, `cdbsql`, `cdbsvcd`, `cdbpkg`, `nosetests`, `sphinx-build` etc.). These scripts will automatically choose the instance directory in the buildout (`CADDOK_DEFAULT` is set in the wrapper script). Note that while you can have multiple instances in one buildout, the scripts are set up to connect the last instance created. While one can use the usual CIM Database command line options to point these scripts (or the unaugmented ones from the platform distribution) to any instance in a buildout or elsewhere, it is recommend to have exactly one instance part in an application development environment to avoid confusion.
3. Create `bin/setpath.bat` or `bin/setpath.sh` scripts that can be sourced or CALLed to set PATH to `bin/`.
4. Create an "egg-link" to the `cs.platform` package in the underlying platform distribution in `develop-eggs` to satisfy `cs.platform` requirements of packages under development (approach and code taken from osc.recipe.syseggs).

For example:

```
C:..\mypackage> buildout -N
... setup instance ...
C:..\mypackage> call bin\setpath.bat
C:..\mypackage> which cdbsql
./bin/cdbsql.exe
C:..\mypackage> cdbsql -v
Connecting database dev_inst@:C:..\mypackage\dev_inst
Encoding IN cp850 OUT cp850
SQL>
```

Note how `cdbsql` is used without further arguments.

3.1.1 Documentation

Documentation sources are stored in the `doc/` directory of the package development tree. For each documentation project or "book" there is a subdirectory, e.g. `admin`, `refman` etc. Inside the book there is a subdirectory for each language, identified by it's ISO 639-1 two-letter code. Inside the language directory is a `src` directory containing the sources of the book.

The build directory for Sphinx will be `...` So after running an HTML build, the `doc` directory will look like:

```
doc/
  refman/
    en/
      src/
        Makefile
        index.rst
        conf.py
        ..
      html/
      doctrees/
```

This structure, sans `src`, goes into the distribution egg as well. See *Writing documentation* (page 19).

3.1.2 Development tools

All tools required for developing packages are provided by the platform: the Python interpreter, the Sphinx documentation system, setuptools/distribute, testing tools, `cdbpkg` and `snapp`, and naturally the CIM Database APIs.

3.2 Modules

Modules are the smallest units of the component architecture and collect all kinds of application content under a common name. During runtime, application content must be available either in the database or in the file-system. To make database content available for packaging, distribution and life-cycle management, it is serialized and written to and read from the file-system by appropriate platform tools.

3.2.1 Database assets

Besides their "on-disk appearance" modules also are objects in the database, that carry an identifier, version, state, description etc. Database items like class definitions, UI elements and many others got a `module_id` field in CIM Database 10. Setting the `module_id` assigns an item to a module object.

To reduce manual work, the system includes discovery tools to automatically collect "unassigned" database content by following the relationship graph of platform the meta-model: for example, starting at a class description, the system finds related UI elements and assigns them to the same module the class belongs to. We expect those tools to be helpful for migration and development.

Database content can be stored to and retrieved from the filesystem. The storage format is JSON (JavaScript Object Notation).

3.3 Bootstrapping and configuration

Bootstrapping, starting up the CIM Database system, is different than before with Release 10. While previous releases loaded the framework and imported application Python code more or less on demand, supported by the `classes.txt` mechanism, release 10 loads all installed modules on startup and checks for consistency with the instance database. This applies to all programs using application level code, i.e. the application server, the `powerscript` command line etc. The exact procedure is described below.

Also new is a configuration mechanism for modules. The mechanism enables:

- setting "options", that control aspects common to all modules; for instance, additional application code hosted in `std-solutions` before is now distributed over the appropriate modules and can be switched on by setting the `"std-solutions"` option.
- module-wise configuration files that carry module-specific settings. These files are *not* meant to be customized.

For the "options" part, two new APIs have been added:

1. `enable_config(<option>)` to be used inside the instance configuration files in `BASE/etc`. The call sets the given option in an internal registry of options. For example in `etc/site.conf`:

```
# site.conf --
...
enable_config("std-solutions")
...
```

   Note that it is unnecessary to import `enable_config` in `*.conf` files.

2. `require_config(<option>)` on the other hand, queries the internal registry for the given option, and is intended to be used inside module code, to protect against errorneous imports. When the option is not set, the call raises an appropriate exception. For example:

```
# thismoduleisforstdsolutionsonly.py --
from cdb.setup import require_config
...
# Sentinel: raise when "std-solutions" is not set
require_config("std-solutions")
...
```

For the configuration part, modules can optionally have module-specific `*.conf` files that are evaluated during bootstrapping. The file `default.conf` is always evaluated. For each "option" set by `enable_config` an additional `<option>.conf` file is evaluated. For example:

```
cs/
    costing/
        default.conf              ..... always
        std-solutions.conf        ..... only if "std-solutions"
        addonsforstd.py
```

Additional code required by an option can be imported in the `<option>.conf` file. In the example above:

```
# std-solutions.conf
import cs.costing.addonsforstd
```

The bootstrap mechanism works as follows:

1. Scan `PYTHONPATH` for all installed packages.
2. Read all installed *packages* from the databases.
3. Compare both sets and their respective versions. When there is a difference, stop startup with error message.
4. Evaluate `CADDOK_BASE/etc/*.conf` (eventually setting options)
5. For each *module* registered (and "activated") in the database:
    1. Import the module (i.e. running the top-level `__init__.py` in the module directory).
    2. Evaluate `default.conf` if present.
    3. For each active option evaluate `<option>.conf` if present.

    Note, that the order of steps is essential here to avoid that "options" influence the module setup in steps 1 and 2.

3.4 Installation of packages

Packages can be installed anywhere, not only in the instance directory (`CADDOK_BASE`). An instance "sees" all packages available in `PYTHONPATH`; package discovery uses the standard `setuptool` APIs to enumerate installed Python packages.

This makes it possible to use different installation strategies and the relevant Python tools to support that. Creating or updating instances synchronizes the instance database with all visibile packages

Possible deployment modes are:

- Platform and instance in different folders; packages go to `CADDOK_BASE/site-packages`
- using `virtualenv` and `buildout` to automate the setup of development and production environments (best practice in the Python community)
- pre-configured distributions with a set of packages in the platform folder
- Any other mechanism to manipulate `PYTHONPATH`

3.5 Master area and package directory

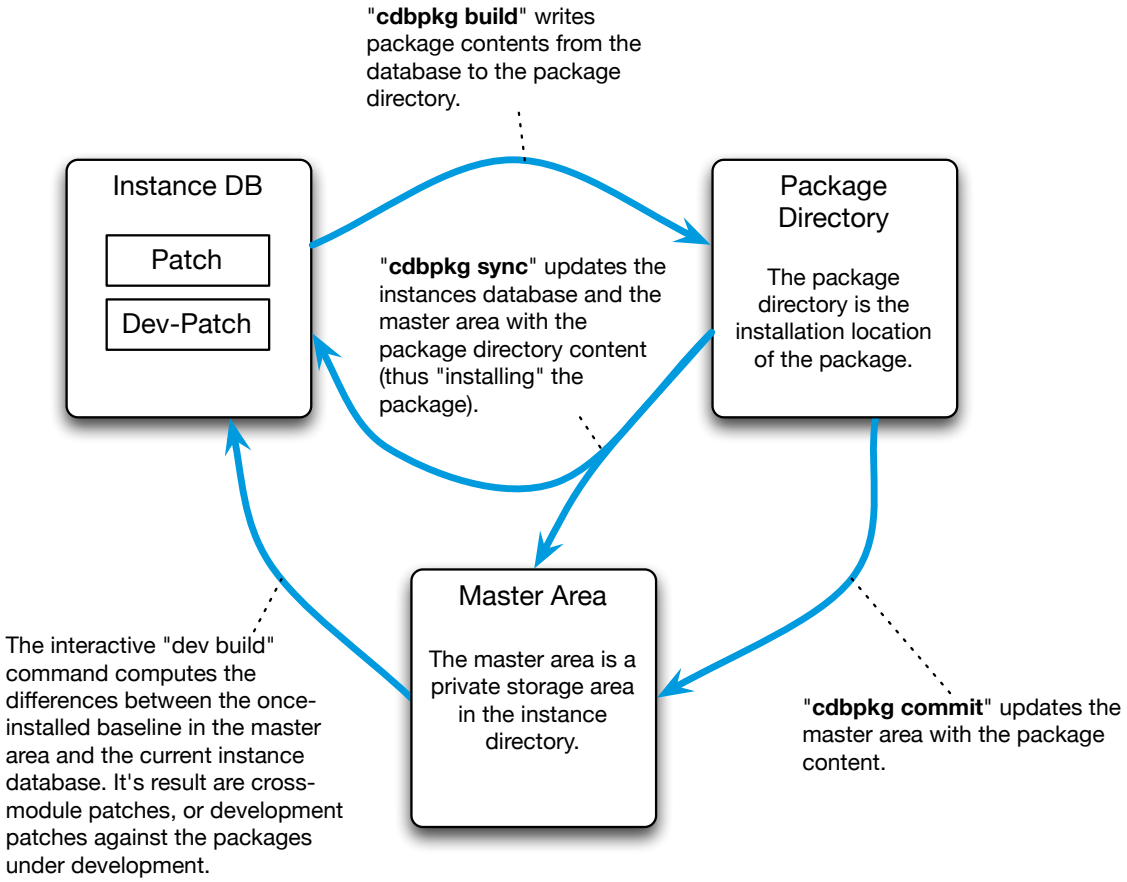

Fig. 2: What `cdbpkg` does

Inside the instance directory, CIM Database holds copies of all packages *as originally installed* in a structure called the "master area". The master area is used to compute differences and patches when packages are developed or customized. Different subcommands of `cdbpkg` move database content between the instance database, the package directory and the master area (see fig. *What cdbpkg does* (page 12)).

CHAPTER 4

Tools

4.1 cdbpkg

The command **cdbpkg** is used while deploying and developing packages to manage the database application content of packages and modules. It is a standard CIM Database tool that runs attached to an instance (and therefore has all the connection options, like `--I` etc.) and has several subcommands. The subcommands are described in the following sections.

4.1.1 sync

Synchronizes the set of packages found in `PYTHONPATH` with the instance database. `sync` is used when installing new packages.

4.1.2 build

Writes package content from the database to the package directory. `build` is used during development to update a package directory under development to prepare the publication of a new increment (intra-team) or a new relases to users.

4.1.3 commit

Updates the master area to reflect a new baseline. `commit` is used during development to create a new master baseline after publishing a new increment of a package. Failure to run `commit` results in wrong patch information displayed in the CIM Database package development user interface.

4.1.4 xliff

Export and import the localizable field data in the XLIFF format. The `xliff` subcommand is used during localization of a package. Typically a stable state is exported via *`--export`* (page 13) and sent to your translation contractor. The resulting translation is later imported with the *`--import`* (page 14) Option. This option updates the database contents. The translations can then be exported into a persistent form with the *build* (page 13) command.

--export pkgs
: Export all localizable fields from the given list of packages in the xliff format.

--import
: Import the localized XLIFF in the given directory into the instance database.

--importdir dirname
: Directory containing the localized .xlf files

--exportdir dirname
: Directory to use for the exports. A subdirectory is created for each package specified. Exporting into the same directory overwrites older .xlf files.

--targetlang lang
: Language specifier for a valid target language. Use ISO style language names. This is encoded in the resulting XLIFF file and any existing translations are loaded from appropriate language fields.

--sourcelang lang
: Source language to use for the export, defaults to de.

--job job_id
: Add a job ID to the generated XLIFF files. The Job ID may be any valid string useable as an XLIFF job ID. Job IDs can be used to track those files through workflows.

--contact-name name
: Add name of a contact person to the generated XLIFF files.

--contact-phone phonenumber
: Add a contact phone number to the generated XLIFF files.

--contact-email email
: Add a contact email address to the generated XLIFF files.

--copy-schema
: Adds a copy of the XLIFF XSD to the export directory.

--verbose
: Provide additional log messages during operation.

Export example:

```
cdbpkg xliff --export --targetlang en --exportdir en cs.platform
<lots of output>
ls en/cs.platform/*.xlf
```

Import example:

```
cdbpkg xliff --import --importdir en/cs.platform
<lots of output>
```

Note: You can exclude fields from extraction, by listing them in a file called blacklist.txt in the current working directory. On lowercased field name per line. This may be needed if your customized schema contains fields that look like localizable fields (names with prefixed or suffixed language codes), but aren't.

See also:

XLIFF FAQ

https://wiki.oasis-open.org/xliff/FAQ

4.2 snapp

The **snapp** program is the extensible automation tool for package development tasks and instance management. `snapp` mainly calls and orchestrates other tools to perform the task at hand. Most `snapp` subcommands expect to run inside a package development directory and rely on the standard Python packaging and deployment tools "setuptools" and "buildout".

For a detailed selection catalog run `snapp --help` or `snapp <subcommand> --help` respectively.

`snapp` will show the commands it calls in it's output, prefixed with `snapp: ...`. The tasks performed by `snapp` can also be performed by issuing those commands manually on the command line.

4.2.1 quickstart

The subcommand `quickstart` generates or updates the configurations files in a package directory: `buildout.cfg`, `README.rst` and `setup.py`. Existing files are not overwritten, unless the `-x` command line option is used.

4.2.2 buildout and upgrade

`snapp buildout` and `snapp upgrade` use Python's buildout to create or update a development instance.

4.2.3 clean

`snapp clean` cleans up a development workspace by removing output directories created by `buildout`, `setuptools` and `Sphinx`.

4.2.4 test

Runs a package's tests by calling **nosetests** on the buildout instance.

4.2.5 doc

Builds a package's documentation by translating all documentation sets given in the `docsets` option of `setup.py`.

4.2.6 upload

Creates an Egg of the package under development and uploads it to the given package repository.

4.2.7 makedocset

Creates a new subdirectory for a documentation set, generating the required subdirectories and configuration files. Existing files are not overwritten, unless the `-x` command line option is used. To add the new documentation set to the package distribution, it still has to be added to the packages's `setup.py`.

4.2.8 maketranslated

Creates a subdirectory for a documentation set translation, generating the required subdirectories and configuration files.

Translated documentation has a `doclink.txt` which refers to the original language documentation and a set of directories called `locale/`*`LANG`*`/LC_MESSAGES` which will contain the translated texts.

To add the new translated documentation set to the package distribution, it still has to be added to the package's `setup.py`. In addition, you should edit the `src/conf.py` file in the new set to add an appropriate translated title to the documentation set.

4.2.9 sync_translation

Prepares the `Sphinx` translation workflow. This synchronizes the .rst texts from the translation source (see `doclink.txt`) to the target dir. Any changes are copied into the existing translations.

The synchronized src documentation needs to be translated with **`snapp doctranslate`**.

4.2.10 doctranslate

Runs the `Sphinx` translation workflow. This extracts the translatable text from the documentation into GNU PO files. Any changes are merged into the existing translations, so the current state of translation can be seen in the results at `locale/`*`LANG`*`/LC_MESSAGES`.

The translated documentation needs to be built with **`snapp doc`** as usual. This step just updates the sources and translated files.

4.2.11 translate

Convenience tool for managing localization for a package. When run with the mode `export` this extracts the localizable texts from both the database and the documentation sets into the export directory. In mode `import` it imports the localized results from a previous export into the package, updating the database and the docset.

4.3 mpq

The **`mpq`** program (Morepath Query Tool) lets you query the underlying morepath configuration database for interesting things. Morepath is the webframewok used by CONTACT Elements. You can for example query all path declarations, all internal views or for all POST views.

```
> mpq path
App: <class 'cs.platform.web.static.publisher.PublisherApp'>
  File "[...]\cdb\python\cs\platform\web\static\publisher.py", line 38
  @PublisherApp.path(path="{name}/{version}", model=Resource, absorb=True)

App: <class 'cs.platform.web.rest.app.CollectionApp'>
  File "[...]\cdb\python\cs\platform\web\rest\app.py", line 34
  @CollectionApp.path(path='', model=RestClassesModel)

File "[...]\cdb\python\cs\platform\web\rest\generic\main.py", line 64
  @CollectionApp.mount(app=App, path="{rest_name}")
[...]
```

```
> mpq view internal=True
App: <class 'cs.restgenericfixture.RelshipParentApp'>
  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 141
```

(continues on next page)

(continued from previous page)

```
  @App.view(model=Object, name="relship-target", internal=True)

  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 150
  @App.view(model=Object, name="base_data", internal=True)

App: <class 'cs.restgenericfixture.RelshipChildApp'>
  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 141
  @App.view(model=Object, name="relship-target", internal=True)

  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 150
  @App.view(model=Object, name="base_data", internal=True)

App: <class 'cs.platform.web.rest.generic.main.App'>
  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 141
  @App.view(model=Object, name="relship-target", internal=True)

  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 150
  @App.view(model=Object, name="base_data", internal=True)
```

```
> mpq view request_method=POST
App: <class 'cs.platform.web.rest.relship.main.RelshipApp'>
  File "[...]\cdb\python\cs\platform\web\rest\relship\main.py", line 357
  @RelshipApp.json(model=RelshipTarget, request_method='POST')

App: <class 'cs.platform.web.rest.generic.main.App'>
  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 58
  @App.json(model=model.ObjectCollection, request_method='POST')

  File "[...]\cdb\python\cs\platform\web\rest\generic\file_view.py", line␣
↪47
  permission=permissions.ReadPermission)

App: <class 'cs.restgenericfixture.RelshipChildApp'>
  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 58
  @App.json(model=model.ObjectCollection, request_method='POST')

  File "[...]\cdb\python\cs\platform\web\rest\generic\file_view.py", line␣
↪47
  permission=permissions.ReadPermission)

App: <class 'cs.restgenericfixture.RelshipParentApp'>
  File "[...]\cdb\python\cs\platform\web\rest\generic\view.py", line 58
  @App.json(model=model.ObjectCollection, request_method='POST')

  File "[...]\cdb\python\cs\platform\web\rest\generic\file_view.py", line␣
↪47
  permission=permissions.ReadPermission)
```

4.4 mppt

The **mppt** program (Morepath Path Tool) lets you query the underlying morepath configuration database for all paths generated by a Morepath app, including points of definition. Morepath is the webframewok used by CONTACT Elements. The programm generates a report with colums path, directive, filename, lineno, model, permission, view_name, request_method and extra_predicates. The Output format is either text or a CSV format.

```
> mppt --format=csv report.csv
```

or

```
> mppt --format=text report.txt
> head report.txt
/                                                                              path  File
 ↪"[...]\cdb\python\cs\platform\web\root\main.py", line 30
html  File "[...]\cdb\python\cs\platform\web\root\main.py", line 35
/about                                                                         mount File
 ↪"[...]\cs.web\cs\web\components\about\main.py", line 49
/about/                                                                        path  File
 ↪"[...]\cs.web\cs\web\components\base\main.py", line 190
html  File "[...]\cs.web\cs\web\components\base\main.py", line 250
name=account_menu_items                                                        view  File
 ↪"[...]\cs.web\cs\web\components\base\main.py", line 472
name=additional_head                                                           html  File
 ↪"[...]\cs.web\cs\web\components\base\main.py", line 349
name=additional_navbar_items                                                   view  File
 ↪"[...]\cs.web\cs\web\components\base\main.py", line 439
name=app_component                                                             view  File
 ↪"[...]\cs.web\cs\web\components\about\main.py", line 59
name=application_help_id                                                       view  File
 ↪"[...]\cs.web\cs\web\components\base\main.py", line 496
```

CHAPTER 5

Writing documentation

Package documentation is organized in *documentation sets*. A documentation set (or *docset*) is a manual, like a user manual or a programming manual. Each package can have multiple docsets. The docsets for a package are declared in the packages's `setup.py` file using the `docsets` options:

```
from cdb.comparch.pkgtools import setup

setup(
  ...
  docsets=[
     "doc/usermanual/en",
     "doc/progman/de (html,pdf)",
     ...
  ],
```

Each item of the `docsets` option consists of a relative path to the location of the documentation. By convention, all docsets are located inside the `doc/` directory, with a subdirectory for each docset and language. The docset path in `setup.py` may be augmented by a comma separated list of documentation formats in parentheses that specifies in what formats the docset is available; the default format is `html`, other format options are `pdf` (Acrobat PDF) and `chm` (Windows Help file). If formats are declared, `html` must be given explicitly, otherwise it is left out.

The docset declarations in `setup.py` are used for different purposes:

- The platform packaging infrastructure automatically collects all content from the `html/` subdirectory of the docset path.
- The packaging infrastructure writes a `docsets.txt` file to the Egg metadata which is used by the documentation portal to discover installed docsets.
- The `snapp` tool uses the list of docsets to automate the documentation build process (using `snap doc`).
- Generates package-wide `docportal.db` and `_docindex` inside the `doc` directory for use in the documentation portal embedded since version 15.1. The file `docportal.db` contains the contents of all books of the package, and `_docindex` provides data for the package-wide search functionality introduced with the documentation portal since version 15.1.

Thus, docsets are packaged and deployed together with the package distribution egg and are automatically picked up by the documentation portal when the package is installed.

Fig. 1: The figure shows four documentation sets as listed in the documentation portal and the `setup.py` declaration leading to this output. The lower two sets offer Acrobat PDF. All sets except the third offer HTML.

5.1 Inside a docset

It is possible to store and deploy arbitrary documentation content using a docset as long as the `html/` subdirectory has an `index.html` file to be used by the documentation portal. It is recommended to use the documentation toolset provided by the platform, which uses the reStructuredText markup language, and the Sphinx documentation generator. The platform provides automation tools, consistent theming etc. for this toolchain.

To create a new docset, the command `snapp makedocset <book> <lang>` should be used. This creates a new docset path, and a Sphinx project in it's `src/` subdirectory populated with a couple of files:

```
> snapp makedocset manual en
snapp makedocset manual en
Creating f:\cdb\trunk\doc\manual\en\src\build.bat ...
Creating f:\cdb\trunk\doc\manual\en\src\conf.py ...
Creating f:\cdb\trunk\doc\manual\en\src\index.rst ...
Creating f:\cdb\trunk\doc\manual\en\src\Makefile ...

Don't forget to add doc/manual/en to the 'docsets' list in setup.py!

>
```

`build.bat` and `Makefile` are used to build documentation from reStructuredText sources. `conf.py` is a Sphinx project configuration file. Those files require the platform for common settings and reused assets.

5.2 conf.py and cdb.sphinxconf

`conf.py` is the configuration file for the Sphinx documentation project. The platform provides the `cdb.sphinxconf` module to simplify and standardize the configuration file. The default template generated by "makedocset" uses `cdb.sphinxconf` to generate a base setup for Sphinx. The content of the generated `conf.py` looks like:

```
from cdb.sphinxconf import configure
from cdb.sphinxconf import kDocCategoryProgramming
configure(globals(),
        title="Writing Package Documentation",
        category=kDocCategoryProgramming)
```

Which is sufficient to build a standard docset with the the look and feel and default settings of the platform.

The arguments for the `configure` call are used as follows:

- **title** will be the headline of the HTML output and the title for the PDF output (if requested)
- **category** is the category under which the documentation portal automatically lists the docset when installed. You should use the constants defined in `cdb.sphinxconf` to categorise the documentation. If there is no suitable category, use a string that contains the name of the new category.

A docset also has a technical name, which is derived from the docset's path (the path component after `doc/`). Like `name`, the docset `language` is automatically derived from the docset path as the second component after `doc/`. `language` is the 2-letter ISO language code for the main language of the documentation set.

`name` and `language` can also be passed as keyword arguments to `configure`.

This meta-data is written by `cdb.sphinxconf` to a file `docset.info` in the `src/` subdirectory and distributed with the docset. The meta-data file is used by the documentation portal.

Overwriting or adding other Sphinx configuration file options should be done *after* the `configure` call. Documentation writers should carefully review what Sphinx and `cdb.shinxconf` are doing with the options in question.

`cdb.sphinxconf` configures the Sphinx project to use the platform documentation standards:

- to use the documentation theme distributed with the platform
- use the Sphinx extension `sphinx.ext.intersphinx` and `sphinx.ext.autodoc`
- intersphinx setup can be simplified by passing `configure` a list of cross-referenced packages using the `intersphinx` keyword argument; this assumes, that cross-referenced packages are checked out, built and located in the same directory as the package under development.
- include shortcut definitions delivered with the platform
- a `html_logo` keyword argument, if given in `configure`, replaces the default CONTACT Elements logo in the html output. It must be the name of an image file (path relative to docset/src) and should have the size 36x36px.
- a `latex_author` keyword argument, if given in `configure`, replaces the default CONTACT Software author on the start page in the pdf output.

For more details consult the source code of `cdb.sphinxconf`.

5.3 Teaser text

In the documentation portal since version 15.1 the docset can contain teaser information to be displayed in the portal. This information will be extracted from the first paragraph of `index.rst` by default. But it can be explicitly specified within `teaser.rst`. Since the `teaser.rst` is not required to be included from another document, a directive `:orphan:` following by a newline should be added at the beginning of this file, in order to avoid warnings during the building process of the documentation.

5.4 Manual builds

A local, non-distributable build of the Sphinx documentation project can be run using the generated `Makefile` (in case GNU Make is available[1]) or `build.bat`. This is usually done while writing and previewing documentation. The tools print help and usage information when called without arguments. The build output is generated in the `_build/<format>` subdirectory below `src/`.

[1] GNU Make is not part of the platform distribution and normally not available on Windows systems.

5.5 Release builds

For "regular" or release builds, `snapp doc` is called inside the package top-level directory where `setup.py` is located. `snapp doc` reads the list of docsets from `setup.py` and automatically builds all requested output formats for all docsets. The set of output formats can be restricted using the `-b` command line option. The set of docsets built can be restricted by passing docset names on the command line (see also usage of `snapp doc`). Regular builds place their outputs in the `html/` subdirectory of the docset directory, resulting in a layout like the following:

```
doc/
  usrman/
    en/
      src/
        conf.py
        index.rst
        ...
      html/
        index.html
        usrman.pdf
        ...
```

The `html/` subtree is the part which gets packaged into and distributed with the package egg file.

Note: `snapp doc` takes the version identifier from the project's `setup.py` field and automatically sets the `version` and `release` configuration options when building the documentation.

5.6 Generating JavaScript documentation

JavaScript APIs are documented in the source files in which these APIs are implemented. If this documentation needs to be included in a docset, `snapp doc` has to generate Restructured Text from the source files. This can be achieved by adding the documentation set to the option `jsdoc` in the applications `setup.py` file.

```
from cdb.comparch.pkgtools import setup

setup(
  ...
  docsets=[
     "doc/usermanual/en",
     "doc/progman/de (html,pdf)",
     ...
  ],
  jsdoc=[
     "doc/progman/en"
  ],
  ...
```

This entry will run `jsdoc` for all web application's defined in the application package and generate the appropriate documentation in `_jsapi`. For details on documenting JavaScript APIs see the `cs.web` developer manual.

CHAPTER 6

Including installers

In addition to the documentation packages, the web portal of CIM Database can provide installation packages for clients to install locally, for example the CIM Database client. This can be achieved by supplying the option `installers` inside the package's `setup.py`, like this:

```
setup(
  ... ,
  installers=["installer1.py", "installers2.py", ... ],
)
```

Consequently, the filepaths listed in this option have to exist relative to the package's root directory. These contain the meta data for each installer, which are

- a category
- the relative filepath inside the package
- an id, which is unique across the installers in this package
- the hash algorithm which shall be used to generate the hash value for the installer file
- a name and a description in each supported language

The name and description are both optional. If no name is supplied, the filename of the installer is displayed instead.

The configuration files have to be structured as follows:

```
from cdb.installerconf import installerCategoryOffice
from cdb.installerconf import configure

installer = {
    "category": installerCategoryOffice,
    "filepath": "filename.msi",
    "id": "an_installer",
    "hash": "md5",
    "info":
    {"de":
        {"name": "Deutscher Name",
        "description": "Eine lange Beschreibung..."
        },
     "en":
```

(continues on next page)

(continued from previous page)

```
        {"name": "English Name",
        "description": "A long description..."
        }
    }
}

configure(installer)
```

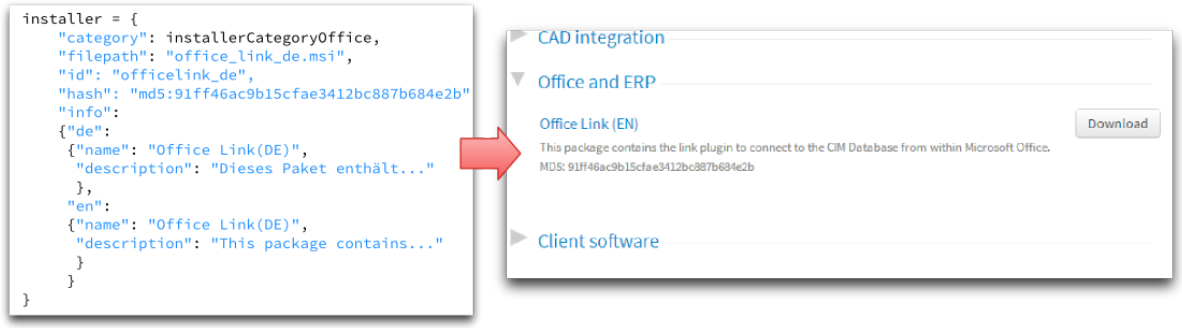

Fig. 1: This figure shows an example output of the installers portal, based on an installer configuration

There is a set of rules that have to be followed:

- Each installer entry in the list of installers can only be linked to one file. So if a client package contains multiple files, they have to be compressed into one archive file
- To prevent a cluttering of category definitions, these are defined at a central location, which is `installerconf.py` inside the cdb platform python module
- Each installer needs to have at least an English description, all other languages are optional.
- The hash algorithm has to be supported by the python hashlib library

The *filepath* attribute of the installer configuration is the relative path to the installer file inside the python egg. A path entry is given in the Unix-like format, using "/" as a separator. It is the responsibility of the package maintainer to determine the output path of the installer file and configure it in the installer configuration. As a result, the installer portal does not work with a development buildout. To test the installer download, an egg has to be packaged and installed into the instance using **`cdbpkg`** or the egg info must be generated using **`powerscript setup.py egg_info`**.

6.1 Excluding installers

It is possible to deny access to the installer portal or hide certain installers from the portal listing. The following variables can be set as environment variables prior to starting the Service-Daemon or inside the configuration file *$CADDOK_BASE*`/etc/cdbsvcd.conf`

To block any access to the portal, the variable `CADDOK_HIDE_INSTALLERPORTAL` has to be set to a non-empty value, e.g. `"True"`. Setting this variable causes the portal to hide all packages, block download links and to show a message, that the installer portal is unavailable.

Hiding specific installers from the portal and blocking their download is done by listing the installer IDs in the variable `CADDOK_HIDE_INSTALLERS`, separated by semicolons. For example:

```
CADDOK_HIDE_INSTALLERS="cs.officelink/officelink_de;cs.officelink/officelink_en;"
```

CHAPTER 7

Translation and internationalization

A package contains content that can be localized or translated into a different language. Examples include:

- GUI labels
- Various messages shown to users
- Documentation

Of course there are a lot more parts to handle for a fully localized package, but those above are the basics.

This chapter assumes you use SVN for version control, but the principle is identical if you use any other version control system, just use the appropriate commands.

7.1 Preparing for localization

On the database side, you have to configure your classes to provide localized fields in the DD for the target languages of all localizable content.

All user visible messages and error messages should be generated from the message catalogs and not be hard coded.

The documentation sets need to be setup for translation via **`snapp maketranslated`**.

7.1.1 Preparing a docset for translation

If a documentation set shall be available in multiple languages, the usual way to do this is by sending the documents to your translation contractor and merge the results of that translation workflow back into the package.

A docset needs to provide the following parts to be translatable:

1. The source text in `.rst` format in the original (untranslated) language
2. (optional) images adapted to the translated language
3. The language neutral images of the original docset
4. Message catalogues containing the translations for the documentation

The **snapp maketranslated** can be used to setup a translation docset for a new language in the following way, as an example this is the `cs.actions` package and the manual *actions_user*, which shall be translated to english.

To do this, we change into the package directory and run the **snapp maketranslated** command for this docset and the source language `de` and the new targetlanguage `en`:

```
cd /D d:\sandbox\cdb10.0\pkgs\cs.actions
.\bin\setpath.bat

snapp maketranslated actions_user de en
snapp: creating 'en' docset at 'doc\actions_user\en'
snapp: svn add doc\actions_user\en
snapp: svn copy doc\actions_user\de\src doc\actions_user\en\src

Please edit doc/actions_user/en/src/conf.py to add a translated title.
Don't forget to add doc/actions_user/en to the 'docsets' list in setup.py!
```

As we work in an SVN controlled repository, `snapp` tries to be helpful and already adds the needed files and directories to our working copy as `svn status` shows.

```
svn status
A       doc\actions_user\en
A       doc\actions_user\en\locale
A       doc\actions_user\en\locale\en
A       doc\actions_user\en\locale\en\LC_MESSAGES
A       doc\actions_user\en\doclink.txt
A  +    doc\actions_user\en\src
```

The `src` is copied over from the original docset via `svn copy` so later updates can more easily track changes. If SVN is not used for the working directory or if the option `--nosvn` is used, the `src` is simply copied.

Now the docset needs to be added to the `setup.py` of the package, so it is found by **snapp doc** and the other tools.

```
docsets=[
    # Add a relative path for each documentation set in this package
    "doc/actions_user/de (html,chm,pdf)",
    "doc/actions_user/en (html,chm,pdf)",
    ],
```

Now open the `src/conf.py` and add a translated title, as it should show up in the documentation portal.

```
from cdb.sphinxconf import configure
from cdb.sphinxconf import kDocCategoryUser

configure(globals(),
    title=u"Actions",
    category=kDocCategoryUser,
    copyright="2015, CONTACT Software")
```

If you already have them, replace all the translatable images in `doc/actions_user/en/src/images` with correctly translated versions.

The docset is now prepared for translation, but if you run **snapp doc** you will see a german document in the english section, as there are no translations provided yet. This will be the next step.

Note: If you have older translations, that do not use this workflow, you need to move them out of the way first. Store them somewhere convenient, as your translation contractor might be able to extract translation memory information from an old translation, which can kickstart a new translation.

7.1.2 Extracting the initial message catalogs for documentation

Initially there are no translations to use, so we need to provide them. To do this, we first extract the translatable messages from the docset.

This can be done with **`snapp doctranslate`**.

Continuing with the example from above, `cs.actions`, we create the initial translation catalogues for the manual *actions_user* in english.

First, build a fresh development snapshot with **`snapp buildout`**, as usual.

In the package directory, than run **`snapp doctranslate`** for the new docset:

```
cs.actions> snapp doctranslate doc/actions_user/en

snapp: [INFO] snapp -d . doc --language en -b gettext doc/actions_user/en
snapp: [INFO] build.bat gettext

Build finished. The message catalogs are in [...]\cs.actions\doc\actions_user\en/
↪locale.
```

As a first part, `snapp` runs a special doc builder called `gettext` to extract message catalogues. The extracted catalogues have the file extension `.pot` and contain untranslated messages and some boilerplate text that is of no further interest.

In the next step, `snapp` converts those POT files into PO files and merges them with any preexisting PO files. As this is the initial run, there are no old PO files, and we just get untranslated files, as you can see from the report.

```
snapp: [INFO] Translation status for actions_user [en]
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_systemaccess.po:0: error: 7␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_masterdata.po:0: error: 42␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\index.po:0: error: 1 translations␣
↪missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_relations.po:0: error: 26␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_lifecycle.po:0: error: 19␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_intro.po:0: error: 8␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_operations.po:0: error: 5␣
↪translations missing
snapp: [INFO] NEW:
  .\doc\actions_user\en\locale\en\LC_MESSAGES\actions_systemaccess.po
  .\doc\actions_user\en\locale\en\LC_MESSAGES\actions_masterdata.po
  .\doc\actions_user\en\locale\en\LC_MESSAGES\index.po
  .\doc\actions_user\en\locale\en\LC_MESSAGES\actions_relations.po
  .\doc\actions_user\en\locale\en\LC_MESSAGES\actions_lifecycle.po
  .\doc\actions_user\en\locale\en\LC_MESSAGES\actions_intro.po
  .\doc\actions_user\en\locale\en\LC_MESSAGES\actions_operations.po
snapp: [INFO] Please make sure the new po/mo files are checked into version␣
↪control.
    Translatable:   108 entries
    Missing:        108 entries
```

The step produced three sets of files. POT contain the untranslated messages in a simple text format. PO files contain the same data as POT, but with already existing translations added and MO is just a binary version of the PO.

So lets follow the advice `snapp` gives and add the new files to our SVN.

```
# add the merged .po files
svn add doc/actions_user/en/locale/en/LC_MESSAGES/*.po
```

It's also a good idea to `svn ignore` the `locale/.doctrees` subdirectory, the `locale/*.pot` files and the the `locale/en/LC_MESSAGES/*.mo` files.

Once done, you can `svn commit` the changes. You now have the basis for sending translation data to your translation contractor.

Note: We do not add the `.pot` files to version control, as those are always completely recreated from the source text.

7.2 Doing a translation

Translation is a little different for the two main parts: documentation and database content.

For documentation you already saw the **`snapp sync_translation`** and **`snapp doctranslate`** command in the previous section, it is the tool also used for iterative translation, not just for the initial steps. For the database content the tool of choice is called **`cdbpkg xliff`**.

Once you understand those two parts, the convenience **`snapp translate`** will be introduced, which combines the two commands and provides a one stop solution.

7.2.1 Translating a docset

Once a docset is prepared for translation, the workflow to get translated documentation follows this simple pattern:

1. Merge any changes from the original language documentation into the translated src dir by running **`snapp sync_translation`**
2. Extract current messages by running **`snapp doctranslate`**
3. Ship `.po` files from `locale/`*`LANG`*`/LC_MESSAGES` to the translation contractor for translation
4. Replace `.po` files in `locale/`*`LANG`*`/LC_MESSAGES` package with the translated `.po` files received from the translation contractor
5. Run **`snapp doctranslate`** to check if all blocks are translated, if not, do 3. again.
6. Run **`snapp doc`** to update the `.mo` files and build the documentation and check if it looks good.
7. Build and upload the new package with translated documentation

Example

A short walkthrough for our example package cs.actions:

Merging the last changes from the original documentation can be done in any way you like. If you use a version control system, like SVN, you can use its features, for example 'svn merge' to move the changes over. If you don't have SVN, you can usually just copy over the files from the original source and just replace the conf.py and translatable images with the right versions.

To simplify the copy / synchronization process of the sources you can run **`snapp sync_translation`**, that copies (using appropriate svn commands if svn controlled) the sources from the origin directory to the translation src directory and removes ommitted files.

```
cs.actions> snapp sync_translation doc/actions_user/en
snapp: [INFO] Syncing docset doc/actions_user/en which originates from doc/actions_
↪user/de
snapp: [INFO] svn cp .\doc/actions_user/de\src\actions_relations.rst .\doc/actions_
↪user/en\src\actions_relations.rst
```

Now, with an updated version, you run **`snapp doctranslate`** to get the up-to-date message blocks extracted.

```
cs.actions> snapp doctranslate doc/actions_users/en
snapp: [INFO] [...]snapp.exe -d . doc --language en -b gettext doc/actions_user/en
snapp: [INFO] build.bat gettext

Build finished. The message catalogs are in [...]\cs.actions\doc\actions_user\en/
↪locale.
snapp: [INFO] Translation status for actions_user [en]
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_systemaccess.po:0: error: 7␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_masterdata.po:0: error: 42␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\index.po:0: error: 1 translations␣
↪missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_relations.po:0: error: 26␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_lifecycle.po:0: error: 19␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_intro.po:0: error: 8␣
↪translations missing
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_operations.po:0: error: 5␣
↪translations missing
snapp: [INFO] Please make sure the new po/mo files are checked into version␣
↪control.
    Translatable:   108 entries
    Missing:        108 entries
```

So, we still have 108 untranslated blocks. Send at least those blocks to your translation contractor. Probably you want to send the complete files for reference.

The files to translate can be found in `doc/actions_user/locale/en/LC_MESSAGES/*.po` in this case, so we send them to the translation contractor and hopefully get a translated version back. PO files contain one line called `msgid` and one line called `msgstr` for every textblock, initially the `msgstr` is empty and it looks like this.

```
#
msgid ""
msgstr ""
"Project-Id-Version: 1.0\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: ..\src\index.rst:4
msgid "Anwenderhandbuch"
msgstr ""
```

Once the translation contractor is done, we might get back the file with this translated text:

```
#
msgid ""
msgstr ""
"Project-Id-Version: 1.0\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
```

(continues on next page)

(continued from previous page)

```
 "Content-Transfer-Encoding: 8bit\n"

 #: ..\src\index.rst:4
 msgid "Anwenderhandbuch"
 msgstr "User Manual"
```

Running **snapp doctranslate** again after adding the translated `.po` files, shows the changes.

```
cs.actions> snapp doctranslate doc/actions_users/en
snapp: [INFO] [...]snapp.exe -d . doc --language en -b gettext doc/actions_user/en
snapp: [INFO] build.bat gettext

Build finished. The message catalogs are in [...]\cs.actions\doc\actions_user\en/
↪locale.
snapp: [INFO] Translation status for actions_user [en]
.\doc\actions_user\en\locale\en\LC_MESSAGES\actions_systemaccess.po:0: error: 7␣
↪translations missing
snapp: [INFO] Please make sure the new po/mo files are checked into version␣
↪control.
    Translatable:   108 entries
    Missing:          7 entries
```

Better, only a few translations missing. It seems the translation contractor didn't do a perfect job yet, so we need another round. Once we are satisfied and there are no missing translations anymore, we can package and release the new version of the package as usual.

7.2.2 Translating database content

For configuration database content as used by the component architecture, you can use the following workflow pattern:

1. Run a fresh **snapp buildout** of your development package
2. Run `cdbpkg xliff --export --targetlang en pkgname` to extract the localizable messages from your package
3. Ship the `.xlf` files to the translation contractor for translation
4. Import the localized `.xlf` received from the translation contractor into the database with `cdbpkg xliff --import`
5. Update the package data from the database with **cdbpkg build**
6. If not all messages are localized yet, restart the process at 1.

Example

As before, we will take a look at the `cs.actions` package. First some preparations.

```
cd cs.actions
snapp buildout
[... lots of output ...]
```

Now lets export the current english message catalogs into a directory xliff

```
mkdir xliff
cdbpkg xliff --export --targetlang en cs.actions --exportdir xliff
Processing: cs.actions
ausgaben                        [    5]
browsers                        [    3]
```

(continues on next page)

(continued from previous page)

```
cdb_wflow                          [    3]
cdbdd_field                        [   37]
masken                             [   10]
meldungen                          [    4]
objektstati                        [    6]
switch_tabelle                     [    8]
tabellen                           [    3]
=======================================
cs.actions                         [   79]
Writing Output for cs.actions to xliff\cs.actions\cs.actions.xlf
```

Ok, 79 messages are in the catalog, from a variety of database classes as listed. Lets have a look at a small part of the exported file. The file format is a bit more complex, but the important parts are the `<xliff:trans-unit>` blocks, which contain the source text and the localized text. In the beginning the `<xliff:target>` is empty (or completely missing), after translation it is filled with the translation.

```
<xliff:trans-unit
    id="cs.actions|cdbdd_field|cdb_action|@end_time_plan|label_de|label_en"
    maxwidth="64" size-unit="char">
    <xliff:source xml:space="preserve">Erledigung bis</xliff:source>
    <xliff:target xml:space="preserve" state="new"></xliff:target>
</xliff:trans-unit>
```

This might look like this, after translation (the details vary a bit due to the tools used by the translation contractor).

```
<xliff:trans-unit
    id="cs.actions|cdbdd_field|cdb_action|@end_time_plan|label_de|label_en"
    maxwidth="64" size-unit="char">
    <xliff:source xml:space="preserve">Erledigung bis</xliff:source>
    <xliff:target xml:space="preserve">To be finished until</xliff:target>
</xliff:trans-unit>
```

Once we got the localized `.xlf` files, we can import them back into our database. We just need to point `cdbpkg xliff --import` at the right directory. It finds out the rest automatically from the included metadata.

```
d:\sandbox\cdb10.0\pkgs\cs.actions>cdbpkg xliff --import --importdir=xliff/cs.
↪actions

Running XLIFF Import
Found 1 XLIFF files in xliff/cs.actions
Validating files for XML Schema violations
xliff/cs.actions\cs.actions.xlf OK
Importing 'xliff/cs.actions\cs.actions.xlf'
Package: cs.actions Version: 0.0 Module: cs.actions Source Language: de Target␣
↪Language: en
Found 61 translation units.
Untranslated string for id ...
Untranslated string for id ...
Untranslated string for id ...
Untranslated string for id ...
Untranslated string for id ...
0 translations updated.
5 translations missing or empty.
```

This import changes the database content, so we need to run **`cdbpkg build`** to export the current version.

```
cdbpkg build cs.actions

Setting module content for module cs.actions
Exporting content of module cs.actions
Build time: 1.88100004196
```

If the translation is completed, we can simply build and release the package as usual. If not, we need to repeat the process until all translations are provided.

7.2.3 Translate with `snapp translate`

This command combines the previous two commands into one.

1. Run a fresh **`snapp buildout`** of your package
2. Run `snapp translate export en ./export` to export the localizable content for english to the `export` dir
3. Send the export dir to the translation contractor for localization
4. Import the localized export dir back into your package with `snapp translate import`.
5. Run **`cdbpkg build`** and **`snapp doc`** to update your package data and see if all is okay.
6. If there are untranslated messages, restart the process.
7. Build and upload the new localized package version as usual.

Example

Basically this just combines the two scripts we have seen before into one step.

```
mkdir translate
snapp translate export en translate
[... output from 'cdbpkg xliff --export' ...]
[... output from 'snapp sync_translation' for all docsets ...]
[... output from 'snapp doctranslate' for all docsets ...]

snapp: processing doc/actions_user/en
snapp: cp actions_intro.po -> workdir
snapp: cp actions_lifecycle.po -> workdir
snapp: cp actions_masterdata.po -> workdir
snapp: cp actions_metrics.po -> workdir
snapp: cp actions_operations.po -> workdir
snapp: cp actions_powerreports.po -> workdir
snapp: cp actions_relations.po -> workdir
snapp: cp actions_systemaccess.po -> workdir
snapp: cp index.po -> workdir
```

In the end, you get a directory structure `translate/doc/cs.actions` containing the PO files and `translate/xliff/cs.actions` containing all the XLIFF files.

After those are localized, the import works like before.

```
snapp translate import en translate
[... output from 'cdbpkg xliff --import' ...]
snapp: Checking 'actions_user'
snapp: Found docset at 'doc\actions_user\en'
snapp: Copying actions_intro.po -> LC_MESSAGES
snapp: Copying actions_lifecycle.po -> LC_MESSAGES
snapp: Copying actions_masterdata.po -> LC_MESSAGES
snapp: Copying actions_metrics.po -> LC_MESSAGES
snapp: Copying actions_operations.po -> LC_MESSAGES
snapp: Copying actions_powerreports.po -> LC_MESSAGES
snapp: Copying actions_relations.po -> LC_MESSAGES
snapp: Copying actions_systemaccess.po -> LC_MESSAGES
snapp: Copying index.po -> LC_MESSAGES
[... output of 'snapp doctranslate' ...]
```

Now you need to finally run **cdbpkg build** to update the package files and **cdbpkg doc** to see if the documentation translation is good.

Once those are fine, you can build and release the package as usual.

7.3 Localize CONTACT Software

Localizing CONTACT Software is meant to be performed by CONTACT Partners.

7.3.1 Assumption

The partner of CONTACT Software GmbH creates English XLIFF-files, translates them and integrates the localized files into the software packages.

The software instance, which the partner must create for a new target language, will be called 'target' instance hereafter.

To help to better understand the process we differentiate certain parts. These parts have to be customized in the concrete localization process by the partner.

These parts are:

- The CONTACT-Software package used in this description is called *cs.pcs*.
- The language is *Korean (ko)*.
- The namespace for the target language is called *target*.

7.3.2 Prerequisites

The partner has read the previous chapters of this manual carefully. He has understood the use of the command *cdbpkg* (page 13).

Providing the software

The partner receives the software from CONTACT-Software GmbH exclusively in English.

Available languages

Currently (10.1) the software is prepared for localization in the following languages:

- Chinese,
- French,
- Italian,
- Korean,
- Japanese,
- Polish,
- Portuguese,
- Spanish and
- Turkish.

Therefore at the moment localizations can only be performed in one of these languages.

If another language is needed CONTACT-Software GmbH can provide this via a software release.

Creating a namespace

The partner has created a 'target' instance with the target namespace for *Korean (ko)*.

Register a target language

The partner registered *Korean (ko)* in the 'target' instance as target language.

Please refer to the administration manual *CIM Database Platform: Administration and Configuration* if you want to know how to register the desired target language. Refer to the chapter *Administration* > *Multilingual capability* > *Configuration*.

Note: Only registered target languages will be shown in the GUI.

7.3.3 Providing the target language for the GUI Elements

Preparation

The following steps must be performed before the partner can localize the XLIFF-files

(a) Create a 'requires' relationship between the English 'source' package *cs.pcs* (English) and the 'target' package *target.cs_pcs* (Korean).

(b) Include the modules, which must be localized for the package *target.cs_pcs*, via a 'customizes' relationship into the 'target' package.

This is how the partner can accomplish these steps:

1. The partner creates a 'target' package which contains a 'requires' relationship with the 'source' package.
   a. Give the 'target' package the name *target.cs_pcs*.
   b. Create a 'requires' relationship with the 'source' package *cs.pcs*.

The command to create the 'target' package is:

```
cdbpkg -l ko new target.cs_pcs -r cs.pcs
```

2. The partner creates a 'customize' relationship between the 'target' package and all the modules, which must be localized.
   a. Open the 'target' instance.
   b. In the navigation area navigate to *Administration* > *Packages & Modules* > *Packages*.
   c. Open the tab *Required Modules*.
   d. Add all the modules needed via the relationship type 'customizes'.

Create the XLIFF-files

Now, the partner can create the XLIFF-files

3. The partner exports the English XLIFF-files from the 'source' package *cs.pcs*.

   Refer to the chapter *Translating database content* (page 30), if you need details to export XLIFF-files. The partner uses the command `cdbpkg()` to initiate the export using the following arguments:

```
xliff, --export, --sourcelang, --targetlang and <pkgname>
```

The partner must provide values for the arguments –sourcelang, –targetlang and <pkgname>.

In this example we will use:

- sourcelang=en
- targetlang=ko
- pkgname=cs.pcs

Therefore the export command would be:

```
cdbpkg xliff --export --sourcelang en --targetlang ko cs.pcs
```

4. The partner localizes the exported XLIFF-files and examines the result.

Important: The partner **must not** translate variable's content, such as {NAME} or #NAME#.

- The content between curly brackets or hash (other sets of special characters are valid, too) **must** remain in the source language.
- If you run across attributes containing ISO code (eg: *text_en* with "en" representing the English language): leave the "text" part untouched but change the ISO code to the corresponding code of the target language, eg: *text_ko*.

Integrate localized XLIFF-files into the 'target' instance

The partner must integrate the localized XLIFF-files into the 'target' instance, if the localization result is satisfactory.

(a) Import the XLIFF-files into the 'target' instance.

(b) Import the translated segments into the 'target' package *target.cs_pcs*.

(c) Build the 'target' package *target.cs_pcs* with the localized GUI.

The partner continues as follows:

5. The partner imports the localization into the database of the 'target' instance. The arguments for importing are different from exporting as the import only requires the command 'import' and the target directory.

The import command is:

```
cdbpkg xliff --import --importdir xliff/<targetfolder>
```

For more details refer to the chapter *Tools* (page 13).

When imported successfully the localized data is available in the database of the 'target' instance.

6. The partner imports the segments of the target language from the database into the 'target' package *target.cs_pcs*. The import into the 'target' package can be performed with:

```
cdbpkg build target.cs_pcs
```

This command includes the Korean XLIFF-files into the 'target' package *target.cs_pcs*.

7. The partner builds the 'target' package *target.cs_pcs.<version>.egg*

For this step he uses the following command:

```
python setup.py bdist_egg
```

8. The partner performs the steps 1 to 7 for each affected package allowing him to install the packages individually.

CHAPTER 8

Reference for cs.recipe.instance

`cs.recipe.instance` is a buildout recipe that creates a CIM Database instance (using an SQLite or Oracle database). It is intended to support application development by adding repeatable and automatic instance creation to package development environments controlled by `buildout`.

8.1 Usage

To automatically create an instance for development, add a part using this recipe to your buildout:

```
[buildout]
...
newest = false
parts = dev_inst

[dev_inst]
recipe = cs.recipe.instance
namespace = froobuzz
```

`namespace` is the only required parameter and is the development namespace given to `mkinstance`.

This will do four things:

1. Create an instance directory named after the part name `dev_inst` inside your buildout.
2. Create scripts in `bin/` that wrap all CIM DATABASE commands that require an instance to run (`powerscript`, `cdbsql`, `cdbsvcd`, `cdbpkg`, `nosetests`, `sphinx-build` etc.). These scripts will automatically choose the instance directory in the buildout (`CADDOK_DEFAULT` is set in the wrapper script). Note that while you can have multiple instances in one buildout, the scripts are set up to connect the last instance created. While one can use the usual CIM DATABASE command line options to point these scripts (or the unaugmented ones from the platform distribution) to any instance in a buildout or elsewhere, it is recommend to have exactly one instance part in an application development environment to avoid confusion.
3. Create `bin/setpath.bat` or `bin/setpath` scripts that can be sourced or CALLed to set PATH to `bin/`.
4. Create an "egg-link" to the `cs.platform` package in the underlying platform distribution in `develop-eggs` to satisfy `cs.platform` requirements of packages under development (approach and code taken from osc.recipe.syseggs.

For example:

```
C:..\mypackage> buildout -N
... setup instance ...
C:..\mypackage> call bin\setpath.bat
C:..\mypackage> which cdbsql
./bin/cdbsql.exe
C:..\mypackage> cdbsql -v
Connecting database dev_inst@:C:..\mypackage\dev_inst
Encoding IN cp850 OUT cp850
SQL>
```

Note how `cdbsql` is used without further arguments.

8.2 Re-creating the instance

`cs.recipe.instance` considers the instance created too "precious" to recreate it all the time, esp. since the user may have added content to the instance database and edited configuration files. Thus, the instance directory will not be overwritten once it has been created. To have the instance recreated, one has to remove the instance directory manually an re-run `buildout`.

8.3 Controlling instance creation

One can pass any of the available `mkinstance` options (e.g. see the output `mkinstance --help` and `mkinstance <dbms> --help`).

The following options have defaults set by `cs.recipe.instance`:

- `dbms` defaults to "sqlite" (this is the first argument to `mkinstance`)
- `instance_name` defaults to the name of the part
- `instance_location` defaults to a directory named after the part inside the buildout
- `instances_conf` defaults to 0 (so `mkinstance` will not write to `instances.conf`)
- `sslmode` defaults to 0

For Oracle databases the following defaults are used:

- `ora_dbhost` defaults to `//localhost:1521/xe`, which is the default connector for a local Oracle Express setup
- `ora_dbuser` and `ora_dbpasswd` defaults to a string computed as a simple hash of the installation path
- `ora_syspwd` defaults to "system" which is dumb but easy